Knowledge-Based General Game Playing

Dissertation
zur Erlangung des akademischen Grades
Doktor rerum naturalium (Dr. rer. nat.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Dipl.-Inf. Stephan Schiffel

geboren am 15. 09. 1980 in Sebnitz

Gutachter:	Prof. Michael Thielscher, University of New South Wales
	Prof. Yngvi Björnsson, Háskólinn í Reykjavík
Datum der Verteidigung:	29.07.2011

Abstract

General Game Playing (GGP) is concerned with the development of systems that can play well an arbitrary game solely by being given the rules of the game. This problem is considerably harder than traditional artificial intelligence (AI) game playing. Writing a player for a particular game allows to focus on the design of elaborate strategies and libraries that are specific to this game. Systems able to play arbitrary, previously unknown games cannot be given game-specific knowledge. They rather need to be endowed with high-level cognitive abilities such as general strategic thinking and abstract reasoning. This makes General Game Playing a good example of a challenge problem, which encompasses a variety of AI research areas including knowledge representation and reasoning, heuristic search, planning, and learning.

Two fundamentally different approaches to GGP exist today, we call them *knowledge-free* and *knowledge-based* approach. The knowledge-free approach considers the game rules as a black box used for generating legal moves and successor states of the game. Players using this approach run Monte-Carlo simulations of the game to estimate the value of states of the game. On the other hand, the knowledge-based approach considers the components of a state (e. g., positions of pieces or tokens on a board) and, based on this information, evaluates the potential of a state to lead to a winning state for the player. This requires to automatically derive knowledge about the game that can be used for a meaningful evaluation.

In this thesis, we present a knowledge-based approach to GGP that is implemented in our general game playing system *Fluxplayer*. Fluxplayer has participated in all the international general game playing competitions since 2005. It won the competition in 2006 and has scored top ranks since then. Fluxplayer uses heuristic search with an automatically generated state evaluation function and applies several techniques for constructing search heuristics by the automated analysis of the game.

Our main contributions are

- the automatic construction of heuristic state evaluation functions,
- the automated discovery of game structures,
- an automated system for proving properties of games, and
- detecting symmetries in general games and exploiting them.

Acknowledgments

Over the last few years, many people helped to make this thesis a reality.

I want to thank my adviser Michael Thielscher for introducing me to this wonderful topic. I am grateful for his constant reminder to publish the results and his invaluable support at writing papers and proof-reading them, even when they were only finished at the last minute.

The people at TU-Dresden and especially my colleagues of the Computational Logic group made the last years a very enjoyable time. There was always someone to discuss ideas or problems and give helpful feedback for papers and presentations. Special thanks go to Conrad for many insightful discussions, some of which were even related to this work. Daniel who was always there for discussing new ideas at length and writing papers at night, and Sebastian whose ideas and suggestions on earlier versions of the thesis considerably helped to improve it.

Finally, I am indebted to my wife Ute for bearing with me all this time. Thank you for urging me to finish this thesis and for struggling through all those unintuitive formulas while proof-reading.

Contents

1. Introduction

Computer systems are used nowadays in many different environments, and new applications are developed every day. In more and more of them computers are not just executing a fixed sequence of tasks but have to take decisions based on their perceptions and a goal they have to achieve. Examples for such systems:

- auto pilots in air planes,
- automated parking systems in cars, but also
- automated trading systems in financial markets.

All these systems have in common that they have an overall goal to achieve. For example, flying or driving to a destination without crashing or making high profits with minimum risk. Furthermore, the environment of these systems and the actors in it behave according to certain rules. Some of these rules might change. For example, if the autopilot is deployed on a new model of an air plane that behaves differently or if tax laws or restrictions of financial markets are changed. In these cases the programs in the systems, more specifically, the conditions under which certain decisions are taken, have to be adapted to the changed environment by hand.

Now consider a program that is given the rules of the environment and the goal that must be achieved. This program shall now automatically make the right decisions to reach its goal. Such a program could fly an air plane, park a car and invest money with a profit automatically. Changes of the environment must only be reflected in the rules that are given to the program. Then the program will automatically adapt to the new environment without the need for further human intervention.

General Game Playing (GGP) is concerned with the development of systems that can play well an arbitrary game solely by being given the rules of the game. Every environment that changes according to rules and contains actors with (possibly conflicting) goals can be easily described as a game. Thus, a true general game playing system would be able to solve many problems that today have to be solved with specific programs, such as, the examples named above. Of course, the complex rules for flying an air plane and large financial markets with thousands of traders are difficult to handle – even for today's computer systems. However, even if general game playing systems cannot compete with special purpose systems directly, they can still be valuable for the development of new special purpose systems: For example, we can use GGP to automatically derive a strategy for an actor in an environment and then this strategy can be hard-coded into the system. In the case of the auto-pilot the developers of a new air plane could use a GGP system to derive a good strategy for the auto-pilot. The system can run in a simulated environment of the plane, thus,

eliminating the problem that decisions have to be taken in real-time. Once the GGP systems has found a good strategy, this strategy can be used as the program that controls the auto-pilot.

The idea of having a program playing arbitrary games was first published under the name of "General Game Playing" in 1968 [Pit68]. It was then long forgotten and picked up again by Barney Pell in his dissertation about Metagamer [Pel93]. Today there is a growing international community of researchers in the area. The annual GGP competitions, hosted by Stanford University since 2005 [GLP05], provide a forum for measuring progress and discussing future research directions.

1.1. The GGP Problem

The goal of General Game Playing is to develop a system, called general game player, that is able to automatically play previously unseen games well. The general game player gets the rules of the game describing

- the roles or players,
- the initial state,
- the move options of the players,
- how the game state evolves depending on the moves taken by the players,
- under which conditions the game ends, and
- the conditions for winning the game.

The system has only limited time to "think" about the rules before the game begins. Then, the player is repeatedly asked to submit a move and told about the decisions of the other players until the game ends. Typically, the time limits for the analysis of the game and submitting moves are too small to search the whole state space for a winning strategy. Thus, decisions have to be taken without knowing all consequences of the decisions in advance.

The original motivation for developing game playing programs was that winning games is connected to exhibiting intelligent behaviour. Thus, research in computer game playing was supposed to lead to artificial intelligence. Unfortunately, success in particular games, such as Chess, has shown that often the intelligence is exhibited by the programmers as opposed to the computer program. Writing a player for a particular game allows to focus on the design of elaborate strategies and libraries that are specific to this game. For example, the strategy of Deep Blue [Mor97], the first program to beat a human world champion in Chess, is composed of thousands of features that were hand-selected by chess experts. The game of Checkers was solved with the help of a large library of Checkers endgames [SBB+07]. Dozens of computers were running for years to produce this library, which is only applicable to Checkers. In contrast, the GGP principle is to give the rules of the game to the program instead of giving them to the programmer. Thus, the player has to do the game analysis on its own. This new problem is considerably harder than traditional AI game playing.

Systems able to play arbitrary, previously unknown games cannot be given game-specific knowledge. They rather need to be endowed with high-level cognitive

abilities such as general strategic thinking and abstract reasoning. This makes General Game Playing a good example of a challenge problem, which encompasses a variety of artificial intelligence (AI) research areas including

- knowledge representation and reasoning,
- heuristic search,
- planning, and
- learning.

In this way, General Game Playing also revives some of the hopes that were initially raised for game playing computers: To develop human-level AI [Sha50].

1.2. Contributions

Most game playing programs use some form of game tree search together with a state evaluation function or heuristics. That means, players compute the states that will be reached by executing certain moves and try to evaluate those states according to their potential to lead to a winning state for this player. Based on this evaluation, a player chooses the move that seems most advantageous to him.

The main problem of General Game Playing is that we cannot predefine an evaluation function for assessing game states for all games but have to automatically derive it for the game at hand. Two basic kinds of state evaluation functions are used in today's general game systems. We call them *knowledge-free* and *knowledge-based* approach.

The knowledge-free approach uses Monte-Carlo simulations of the game to estimate the value of a state. That means, random simulations of the game are played to see how likely it is for the player to win the game starting in a specific state. The approach is called knowledge-free because no further knowledge about the structure of the game states is necessary. The only information that is needed is given directly by the game rules:

- which moves the players can take,
- what the resulting state is when certain moves are taken, and
- who wins in a certain terminal state.

In contrast to the knowledge-free approach, the knowledge-based approach considers the components of a state (e. g., positions of pieces or tokens on a board) and, based on this information, evaluates the potential of a state to lead to a winning state for the player.

Although knowledge-free players seem to dominate the general game playing competitions lately, we believe that automatically acquiring knowledge about the game is the key to the success of general game playing systems. The recent introduction of heuristics to guide the Monte-Carlo simulations in systems like Cadiaplayer [FB10] supports our belief. Therefore, in this thesis, we focus on approaches that automatically acquire and use information about a game to improve state evaluation functions and, thus, the performance of general game players.

In this thesis, we present an approach to General Game Playing that is implemented in our general game playing system *Fluxplayer*. Fluxplayer has participated in all the international general game playing competitions since 2005. It won the competition in 2006 and has scored top ranks since then. In Fluxplayer, we use a knowledge-based approach including a heuristic search method for evaluating states and several techniques for constructing search heuristics by the automated analysis of the game.

The remainder of the thesis is organised as follows: In Chapter 2, we will give an introduction to the set up used in the General Game Playing competitions. We will introduce the Game Description Language, which is used to formally describe the rules of the game. Furthermore, we will present other preliminaries for the remaining chapters including Fuzzy Logic and Answer Set Programs. In the remaining chapters, we will present own research starting with the structure of Fluxplayer (Chapter 3) and an overview of the search algorithms that we use for the different classes of games (Chapter 4). Our main contributions are:

Semantics of the Game Description Language (GDL) In Section 2.14 we define a formal semantics of the game description language GDL as a state transition system. This semantics is used in the remainder of the work for formal definitions and proofs.

State Evaluation Function We develop a method to construct effective state evaluation functions for general games in Chapter 5. Our state evaluation function is constructed directly from the rules of the game and does not require expensive learning. Furthermore, we develop methods to automatically find structures in the game, such as game boards, quantities, and, order relations. We show how these discovered structures can be used to improve the quality of our state evaluation functions.

Distance Estimates In Chapter 6, we present an algorithm to compute admissible estimates for the number of steps needed to fulfil atomic game properties. Again, these distance estimates can be used to improve state evaluation functions.

Proving Properties More knowledge about the game can be extracted by hypothesising and proving state invariants. In Chapter 7, we present a method for this, based on Answer Set Programming. The obtained knowledge can be used for the state evaluation function and for selecting an appropriate strategy for the game.

Symmetry Detection Symmetries, such as symmetric roles or symmetries of the board, occur in many games. Exploiting symmetries can greatly reduce the search space as well as the complexity of other game analysis. In Chapter 8, we develop a sound method for detecting and exploiting symmetries in general games.

In Chapter 9, we discuss related work. We conclude the thesis with a discussion of the results and suggestions for future work in Chapter 10.

2. Preliminaries

In this chapter, we will introduce several topics that are necessary for understanding the remaining chapters. We start with defining what a game is and how to formally describe games with the Game Description Language. Furthermore, we present the classes of games that we will consider in this thesis. We will also introduce Fuzzy Logics which is used later in the thesis for generating state evaluation functions in Chapter 5. Finally, we introduce the Answer Set Programming paradigm which we use for proving game properties in Chapter 7.

2.1. Games

Encyclopedia Britannica defines a game as "*a universal form of recreation generally including any activity engaged in for diversion or amusement and often establishing a situation that involves a contest or rivalry.*" [Bri11]. Clearly, this definition leaves a lot of room for interpretation.

In this work, we restrict ourselves to games that can be modelled as finite state machines. That means, we only consider games with a fixed finite number of players and finitely many different states. The transitions between states depend on the actions (sometimes called moves) of the players. Effects of actions occur instantaneously, that means, actions have no duration. For each state transition, every player has to select one action. Games with alternating moves, such as Chess or Checkers, where only one of the players moves at a time, can be modelled by introducing a *noop* action, that is, an action that has no effect. Players whose turn it is not can only choose this action.

We will now formally define what we consider as a game.

Definition 2.1 (Game)**.** *A game is a state transition system* (R, s_0, T, l, u, g) *over sets of states* $\mathcal{S}$ *and actions* $\mathcal{A}$ *with*

- R*, a finite set of roles;*
- $s_0 \in \mathcal{S}$*, the initial state of the game;*
- $T \subseteq \mathcal{S}$*, the set of terminal states;*
- $l : R \times \mathcal{A} \times \mathcal{S}$*, the legality relation;*
- $u : (R \to \mathcal{A}) \times \mathcal{S} \to \mathcal{S}$*, the transition or update function;*
- $g : R \times \mathcal{S} \to \mathbb{N}$*, the reward or goal function.*

The game starts in some fixed initial state s_0. In every non-terminal state $s \in \mathcal{S}$, each player $r_i \in R$ has to choose one of its legal moves, i. e., an action a_i such

that $l(r_i, a_i, s)$ holds. The successor state $s' = u(\{r_1 \mapsto a_1, \ldots, r_n \mapsto a_n\}, s)$ depends on the current state s and the *joint action* of all players. The joint action is a function $A : R \to \mathcal{A}$ that specifies for each player $r_i \in R$ which action $a_i = A(r_i)$ this player selects. Finally, when a terminal state is reached, each player r gets his reward $g(r, s)$. The goal of each player should be to maximize its own reward.

In the remainder of this thesis, we will consider games over a set of ground terms, such that roles and actions of the game are ground terms and states of the game are finite sets of ground terms:

Definition 2.2 (Game Over a Set of Ground Terms)**.** *Let Σ be a set of ground terms and 2^Σ be the set of finite subsets of Σ. A game over a set of ground terms Σ is a game (R, s_0, T, l, u, g) with*

- *states $\mathcal{S} = 2^\Sigma$,*
- *actions $\mathcal{A} = \Sigma$, and*
- *roles $R = \Sigma$.*

With the definition above, the states of a game are just defined as finite sets of ground terms. Not all of these states can occur in an actual match of the game. Some of the states are not reachable from the initial state by executing legal actions. We define the notion of a legal joint action and the set of reachable states of a game as follows.

Definition 2.3 (Legal Joint Action)**.** *Let (R, s_0, T, l, u, g) be a game with states $\mathcal{S}$ and actions $\mathcal{A}$. We call a joint action $A \in \mathcal{A}$ a* legal joint action *in state $s \in \mathcal{S}$ if and only if the action $A(r)$ of every role $r \in R$ is legal in s. We define the following abbreviation to denote that A is a legal joint action in s:*

$$l(A, s) \stackrel{\text{def}}{=} (\forall r \in R) l(r, A(r), s)$$

Definition 2.4 (Reachable States)**.** *Let (R, s_0, T, l, u, g) be a game with states $\mathcal{S}$ and actions $\mathcal{A}$. The set of reachable states $\mathcal{S}_r$ of the game is the smallest set such that*

- *$s_0 \in \mathcal{S}_r$, and*
- *for all $s \in \mathcal{S}$ and joint actions $A : R \to \mathcal{A}$, if $s \in \mathcal{S}_r$, $s \notin T$ and $l(A, s)$ then $u(A, s) \in \mathcal{S}_r$.*

Furthermore, a state $s \in \mathcal{S}_r$ is called finitely reachable if it is reachable from the initial state s_0 by a finite sequence of joint actions, that is, there are $A_1, \ldots, A_n$ such that $s = u(A_n, u(A_{n-1}, \ldots u(A_0, s_0) \ldots))$.

Informally, the definition says that the initial state s_0 is reachable. Furthermore, every state $u(A, s)$ is reachable if it is the result of executing a legal joint action A in a reachable and non-terminal state s. Note that in games that always end after finitely many steps, all reachable states are also finitely reachable.

The observant reader may notice that the above definition of a game (Definition 2.1) does not allow games with non-deterministic actions, such as rolling of dice. Until now, the General Game Playing competitions only considered a subset of games that adhere to the following three restrictions:

- deterministic actions,
- complete information, and
- finiteness.

Recently, Thielscher [Thi10] introduced a language for describing games with incomplete information and elements of chance. However, until now there is no large enough selection of games nor other systems for comparison that use this language. Thus, in this thesis we only consider finite deterministic games with complete information. We will explain these restrictions and their consequences in the next sections.

2.1.1. Deterministic Actions

The state transitions of a game are solely determined by the actions that are selected. Hence, effects of the actions only depend on the state in which the actions are executed but not on some random element.

Games with an element of chance, e. g., rolling of dice, can in principle easily be modelled by introducing an additional player. This player selects actions (e. g., rolling a dice with a predetermined outcome) randomly according to some random distribution. However, according to the general game playing protocol (see Section 3.2), a player does not know if some other player plays randomly. Thus, players cannot take the random distribution of the selected actions into account.

2.1.2. Complete Information

By complete information we mean that the rules of the game and the current state of the game are completely known by every player at all times. In particular, there is no information asymmetry like in card games, where typically one player knows his own cards, but not the cards of the opponents. This restriction actually excludes a large class of games. However there are still many interesting games that fall into this class. Some of these were or are still considered difficult for computer players, e. g., Chess, Checkers or Go.

The only information that is not known to the players is the strategy of the opponents, that is, which actions the opponents will select in a state. However, the players do observe the actions that have been selected by other players in previous states. There are other notions of incomplete information, e. g., incomplete information about the effects of actions or about the objectives of the other players. In this thesis, we will not consider any of these kinds of incomplete information.

2.1.3. Finiteness

A game in our sense is finite in different aspects. First, we only consider games with finitely many roles, actions and states, as mentioned in Definition 2.1. The second restriction is that we only consider games that end after finitely many

steps. That means, every sequence of state transitions that starts in the initial state of the game will reach a terminal state after finitely many steps. This excludes games were players can choose between an infinite number of moves, for example, betting money, if the upper bound on the amount is not known in advance. It also excludes games in which the maximal number of players is not known in advance. Finally, games are excluded that can go on forever, e. g., playing rock/paper/scissors until one of the player wins[1]. A game that can go on forever, is typically transformed into a finite game by ending the game in a draw after a fixed number of steps, in case no natural end was reached before.

2.2. Game Description Language (GDL)

In the GGP community, the Game Description Language [LHH+08] (GDL) is most widely accepted for encoding game rules. The GDL was developed by the Stanford Logic Group and is used for the annual GGP competitions. This language is suitable for describing finite and deterministic n-player games ($n \geq 1$) with complete information, as defined in Section 2.1. GDL is purely axiomatic, so that no prior knowledge (e. g., of geometry or arithmetics) is assumed.

Although state transition systems provide an axiomatisation of games, describing and communicating games directly as a state transition system is infeasible for all but the smallest games. Even for simple games, such as Tic-Tac-Toe, the number of states are in the thousands. Enumerating all states of larger games is not possible with todays computers.

Instead, GDL uses a a modular representation of the states and actions of a game to encode the game rules. For the purpose of a compact encoding, states need to be composed of atomic properties. We call these properties fluents. In GDL, the fluents are encoded as terms, for example, `cell(X,Y,P)` representing that `P` is the content of square `(X,Y)` on a chess board. A game state is a finite set of such fluents. Thus, the number of states that can be described is exponential in the number of terms used for the game description. The actions are represented by terms, as well. For example, the term `move(U,V,X,Y)` could denote the action that moves the piece on square `(U,V)` to `(X,Y)`.

2.2.1. General GDL Syntax

GDL is based on the standard syntax of logic programs [Llo87], including negation. That means, a game description in GDL is a logic program according to the following definition:

Definition 2.5 (Logic program)**.**

- *A* term *is either a variable, or a function symbol applied to terms that are the arguments of the function. A* constant *is a function symbol with no argument.*

[1] Both players could choose rock all the time and the game would last forever.

- *An* atom *is a predicate symbol applied to terms as arguments.*
- *A* literal *is an atom or its negation.*
- *A* clause *is an implication* $h \Leftarrow b_1 \wedge \ldots \wedge b_n$ *where the* head h *is an atom and the* body $b_1 \wedge \ldots \wedge b_n$ *is a conjunction of literals (with* $n \geq 0$*).*
- *A* logic program *is a set of clauses.*

In GGP competitions, game descriptions are communicated to the players in the syntax of the Knowledge Interchange Format (KIF) [Gen98]. However, we believe that Prolog syntax is easier to read for humans. Thus, we use prolog syntax in the remainder of this thesis. We adopt the Prolog convention according to which variables are denoted by uppercase letters and predicate and function symbols start with a lowercase letter. We also write the inverse implication $\Leftarrow$ as `:-` and indicate the end of each rule with a dot. Clauses with an empty body (e. g., $h \Leftarrow$) are written without the inverse implication as prolog facts (e. g., `h.`).

2.2.2. GDL Keywords

As a tailor-made specification language, GDL uses the following pre-defined predicate symbols with the respective informal meaning:

`role(R)`
: `R` is a player or role in the game.

`init(P)`
: `P` holds in the initial state.

`true(P)`
: `P` holds in the current state.

`legal(R,M)`
: Player `R` has legal move `M` in the current state, i. e., `R` is allowed to execute move `M`.

`does(R,M)`
: Player `R` does move `M` in the current state.

`next(P)`
: `P` holds in the next state.

`terminal`
: The current state is terminal.

`goal(R,N)`
: Player `R` gets goal value `N` in the current state if the current state is terminal. By convention, the goal value is a natural number between 0 and 100, where 0 means loosing and 100 means winning the game.

`distinct(X,Y)`
: The terms `X` and `Y` are syntactically different.

As an example, Figure 2.1 shows the complete GDL description for the game Tic-Tac-Toe. Tic-Tac-Toe has two players: `xplayer` and `oplayer`, as defined in line 1. The game is played on a 3 by 3 board, which is encoded by the ternary function symbol `cell` (lines 2–6). The first and second argument encode the coordinates of the board location and the third argument stands for the content

```
role(xplayer). role(oplayer).
init(cell(a,1,blank)). init(cell(a,2,blank)).
init(cell(a,3,blank)). init(cell(b,1,blank)).
init(cell(b,2,blank)). init(cell(b,3,blank)).
init(cell(c,1,blank)). init(cell(c,2,blank)).
init(cell(c,3,blank)).
init(control(xplayer)).

legal(P, mark(X,Y)) :-
    true(control(P)), true(cell(X,Y,blank)).
legal(P,noop) :- role(P), not true(control(P)).

next(control(oplayer)) :- true(control(xplayer)).
next(control(xplayer)) :- true(control(oplayer)).
next(cell(M,N,x)) :- does(xplayer,mark(M,N)).
next(cell(M,N,o)) :- does(oplayer,mark(M,N)).
next(cell(M,N,C)) :- true(cell(M,N,C)),
    does(P,mark(X,Y)), distinct(X,M).
next(cell(M,N,C)) :- true(cell(M,N,C)),
    does(P,mark(X,Y)), distinct(Y,N).

terminal :- line(x).
terminal :- line(o).
terminal :- not open.

line(P) :- true(cell(a,Y,P)),
    true(cell(b,Y,P)), true(cell(c,Y,P)).
line(P) :- true(cell(X,1,P)),
    true(cell(X,2,P)), true(cell(X,3,P)).
line(P) :- true(cell(a,1,P)),
    true(cell(b,2,P)), true(cell(c,3,P)).
line(P) :- true(cell(a,3,P)),
    true(cell(b,2,P)), true(cell(c,1,P)).

open :- true(cell(X,Y,blank)).

goal(xplayer,100) :- line(x).
goal(xplayer,50) :- not line(x), not line(o).
goal(xplayer,0) :- line(o).
goal(oplayer,100) :- line(o).
goal(oplayer,50) :- not line(x), not line(o).
goal(oplayer,0) :- line(x).
```

Figure 2.1.: The rules of Tic-Tac-Toe in GDL.

of the cell, which is `blank` initially. Since the game is turn-taking, the player whose turn it is has to be specified. This is done using the function `control` in line 7.

In turn, each player places one of his markers on the board. The legal moves of the players are encoded using the keyword **`legal`** in terms of conditions that hold in the current state. The rule in lines 9–10 states that a player, whose turn it is (denoted by `true(control(P))`), can mark any cell that is currently blank (denoted by `true(cell(X,Y,blank))`). The other player can only do a `noop` action (line 11).

The successor state is defined using the keyword **`next`** in terms of the current state and the moves of the players. The first two **`next`**-rules (lines 13–14) define whose turn it is in the successor state. The changes to the game board done by a move are defined in lines 15–16. For example, the rule starting in line 15 says that cell `(M,N)` contains the marker `x` after `xplayer` marked the cell. For a complete description of the successor state, it is also necessary to define which fluents of the state stay unchanged. The rules in lines 17–20 state that the content of a cell is still valid in the next state if a different cell was marked.

The game ends if any player managed to build a line of three of its symbols or if there is no blank cell left. This condition is encoded in lines 22–24 with the help of the additional predicates `line` and `open`. These predicates are not keywords but specific to this game. They are again defined by GDL rules (26–35) in terms of the current state. Finally, when the game is over, the players receive a reward that is defined using the keyword **`goal`** (26–35). In this case, a player receives full points, i. e., 100, if there is a line of three of its symbols, it gets zero points if there is a line of three of the opponents symbols and 50 points in case of a draw.

2.2.3. GDL Restrictions

Players must be able to use the game description for computing legal moves and successor states, among other things. To ensure that these computations can terminate, GDL imposes some general restrictions on the set of clauses. Before we state the restrictions, we first define the notion of a dependency graph for a game description following [LHH+08]:

Definition 2.6 (Dependency Graph)**.** *The* dependency graph *for a set D of clauses is a directed, labeled graph whose nodes are the predicate symbols that occur in D. There is a* positive *edge $p \xrightarrow{+} q$ if D contains a clause $p(\overline{s}) \Leftarrow \ldots \wedge q(\overline{t}) \wedge \ldots$, and a* negative *edge $p \xrightarrow{-} q$ if D contains a clause $p(\overline{s}) \Leftarrow \ldots \wedge \neg q(\overline{t}) \wedge \ldots$.*

We can now define what constitutes a *valid GDL specification*.

Definition 2.7 (Valid GDL Specification)**.** *Let D be a finite set of clauses and G be the dependency graph of D. We call D a* valid GDL specification *if it satisfies the following conditions:*

1. *There are no cycles involving a negative edge in G. This is also known as being* stratified *[ABW87, vG89].*

2. *Each variable in a clause occurs in at least one positive atom in the body. This is also known as being* allowed *[LT86].*
3. *If p and q occur in a cycle in G and D contains a clause*

$$p(s_1, \ldots, s_m) \Leftarrow b_1(\bar{t}_1) \wedge \ldots \wedge q(v_1, \ldots, v_k) \wedge \ldots \wedge b_n(\bar{t}_n)$$

then for every $i \in \{1, \ldots, k\}$,

- *v_i is variable-free, or*
- *v_i is one of $s_1, \ldots, s_m$, or*
- *v_i occurs in some $\bar{t}_j$ $(1 \leq j \leq n)$ such that b_j does not occur in a cycle with p in G.*

This is called recursion restriction *in [LHH+08].*

4. `role` *only appears in the head of clauses that have an empty body.*
5. `init` *only appears as head of clauses and is not connected, in the dependency graph for D, to any of the predicates* `true`, `legal`, `does`, `next`, `terminal`, *and* `goal`.
6. `true` *only appears in the body of clauses.*
7. `does` *only appears in the body of clauses and is not connected, in the dependency graph for D, to any of the predicates* `legal`, `terminal`, *and* `goal`.
8. `next` *only appears as head of clauses.*

Stratified logic programs are known to admit a specific *standard model*. We refer to [ABW87] for details and just mention the following properties:

1. To obtain the standard model, clauses with variables are replaced by their (possibly infinitely many) ground instances.
2. Clauses are interpreted as reverse implications.
3. The standard model is minimal if negation is interpreted as non-derivability. This is also known as the "negation-as-failure" principle [Cla78].

The second and third restriction in Definition 2.7 guarantee that a finite logic program entails a *finite* number of ground atoms via its standard model. This is necessary to enable agents to make effective use of a set of game rules. For example, it would be impractical if a player would have to deal with a state in which an infinite number of ground terms hold (e. g., if there is an infinite number of ground atoms of the form `next`(f) in the model). The remaining restrictions in Definition 2.7 are on the use of the GDL keywords. They ensure that:

- the roles of the game and the initial state are fixed,
- the legal moves, terminality and goalhood only depend on the current state, and
- the successor state only depends on the current state and the moves of each player.

The preceding definitions guarantee game descriptions with which players can reason, that is, compute legal moves, successor states and the outcome of the game. However, these syntactic restrictions are not enough to ensure expedient game descriptions. For example, the restrictions do not ensure that every player always has a legal move or that the outcome of the game is defined for every terminal state. The following definitions for well-formedness of a game description from[LHH+08] solve this problem.

Definition 2.8 (Termination)**.** *A game description in GDL terminates if all infinite sequences of legal moves from the initial state of the game reach a terminal state after a finite number of steps.*

Definition 2.9 (Playability)**.** *A game description in GDL is playable if and only if every role has at least one legal move in every non-terminal state reachable from the initial state.*

Definition 2.10 (Monotonicity)**.** *A game description in GDL is monotonic if and only if every role has exactly one goal value in every state reachable from the initial state, and goal values never decrease.*

Definition 2.11 (Winnability)**.** *A game description in GDL is strongly winnable if and only if, for some role, there is a sequence of individual moves of that role that leads to a terminal state of the game where that roleâs goal value is maximal. A game description in GDL is weakly winnable if and only if, for every role, there is a sequence of joint moves of all roles that leads to a terminal state where that roleâs goal value is maximal.*

Definition 2.12 (Well-formed Game Descriptions)**.** *A game description in GDL is well-formed if it terminates, is monotonic, and is both playable and weakly winnable.*

Many results of this thesis hold for all valid game descriptions regardless of their well-formedness. Especially, monotonicity is not required for any of the results. In fact, the example game description in Figure 2.1 does not obey the monotonicity requirement of well-formed game descriptions.

However, for defining the semantics of a game description in the next section, we find it helpful to restrict ourselves to games that fulfil a weaker version of monotonicity, which we call *outcome definedness*:

Definition 2.13 (Outcome Definedness)**.** *A game description in GDL is outcome defined if and only if every role has exactly one goal value in every terminal state reachable from the initial state.*

Observe, that we only require the goal value for each role to be defined in terminal states of the game.

2.3. Semantics of the GDL

While the other sections in this chapter merely introduce work by others, this section is a summary of own work published in [ST09b].

Based on the concept of the standard model [ABW87], a valid and outcome defined game description can be understood as a state transition system as defined in Definition 2.1. To begin with, any valid game description D in GDL contains a finite signature, i. e., a finite set of function symbols, including constants. This signature implicitly determines a set of ground terms Σ, the symbol base. The roles R, states $\mathcal{S}$ and actions $\mathcal{A}$ of a game are constructed from this set Σ: Roles and actions of the game are ground terms in the game description while states of the game are finite sets of ground terms.

The players and the initial state of a game can be directly determined from the clauses for, respectively, `role` and `init` in D. In order to determine the legal moves, update, termination, and goalhood for any given state, this state has to be encoded first, using the keyword `true`. To this end, for any *finite* subset $s = \{f_1, \ldots, f_n\} \subseteq \Sigma$ of a set of ground terms, the following set of logic program facts encodes the state s as the current state of the game:

$$s^{\texttt{true}} \stackrel{\text{def}}{=} \{\texttt{true}(f_1)\texttt{.},\ \ldots,\ \texttt{true}(f_n)\texttt{.}\}$$

Furthermore, for joint action A, that is a function $A : \{r_1, \ldots, r_n\} \to \mathcal{A}$ that assigns a move to each player $r_1, \ldots, r_n \in R$, the following set of facts encodes A as a joint action in GDL:

$$A^{\texttt{does}} \stackrel{\text{def}}{=} \{\texttt{does}(r_1, A(r_1))\texttt{.},\ \ldots,\ \texttt{does}(r_n, A(r_n))\texttt{.}\}$$

With the help of the definitions of $s^{\texttt{true}}$ and $A^{\texttt{does}}$ we can now define the semantics of a game description:

Definition 2.14 (Semantics of a GDL Specification)**.** *Let D be a valid GDL specification whose signature determines the set of ground terms Σ. Let 2^{Σ} be the set of* finite *subsets of Σ. The* semantics *of D is the game (R, s_0, T, l, u, g) where*

- $\mathcal{A} = \Sigma$ *(the actions);*
- $\mathcal{S} = 2^{\Sigma}$ *(the states);*
- $R = \{r \in \Sigma : D \models \texttt{role}(r)\}$ *(the players);*
- $s_0 = \{p \in \Sigma : D \models \texttt{init}(p)\}$ *(the initial state);*
- $T = \{s \in \mathcal{S} : D \cup s^{\texttt{true}} \models \texttt{terminal}\}$ *(the terminal states);*
- $l = \{(r, a, s) : D \cup s^{\texttt{true}} \models \texttt{legal}(r, a)\}$*, where $r \in R$, $a \in \mathcal{A}$, and $s \in \mathcal{S}$ (the legality relation);*
- $u(A, s) = \{f \in \Sigma : D \cup s^{\texttt{true}} \cup A^{\texttt{does}} \models \texttt{next}(f)\}$*, for all $A : (R \to \mathcal{A})$ and $s \in \mathcal{S}$ (the update function);*
- $g(r, s) = n$ *iff* $G \cup s^{\texttt{true}} \models \texttt{goal}(r, n)$*, for all $r \in R$, $n \in \mathbb{N}$, and $s \in \mathcal{S}$ (the goal function).*

This definition provides a formal semantics by which a GDL description is interpreted as an abstract n-player game.

2.4. Fuzzy Logic

In Chapter 5, we will present our state evaluation function. This function evaluates states against the conditions for winning the game that are given by the `goal` rules of the game description. We will use fuzzy logic for this evaluation. Therefore, we recapitulate some definitions for fuzzy logic from [KGK95] in this section.

Fuzzy logic is a multi-valued logic. Instead of the default truth values "true" and "false" of binary logics, fuzzy logic has truth values in the interval $[0, 1]$ of the real values. In fuzzy logic, a truth value denotes a degree of truth of a formula, where 0 stands for false, 1 stands for true, and the remaining values stand for "true to a certain degree". Fuzzy logics are used for approximate reasoning. For example, the sentence "The sky is blue today." may not be entirely true because there are some clouds or because the sky is not blue the entire day. Still, it might be true for most of the day. Thus, we could associate a truth value between 0 and 1 reflecting the degree of truth of the sentence, e. g., the percentage of the time the sky is blue. For evaluation whether a state s of a game is advantageous for our player, we are interested in the degree of truth of the sentence "the state s is near to a goal state".

Formally, fuzzy logics are defined in terms of fuzzy sets:

Definition 2.15 (Fuzzy set)**.** *Let X be a set of objects. A fuzzy set μ of X is a function $\mu : X \to [0, 1]$, which associates each object in X a real number in the interval $[0, 1]$ representing the "grade of membership" of x in the fuzzy set.*

For example, the fuzzy set μ may contain all states of a game that are near to a goal state for a certain role. The function $\mu(s)$ associates a value between 0 and 1 to each state s, such that the value represents the grade of membership of s in μ. Thus, the value $\mu(s)$ denotes how near s is to a goal state.

We define the three operations complement ($\neg$), intersection ($\wedge$), and union ($\vee$) on fuzzy sets in the following paragraphs.

Complement The complement $\neg\mu$ of a fuzzy set μ is defined elementwise using a negation function n:

$$(\neg\mu)(x) = n(x)$$

Definition 2.16 (Negation function)**.** *A negation function n is a function $n : [0, 1] \to [0, 1]$ with the following properties:*

- $n(0) = 1$
- $n(1) = 0$
- $a \leq b \supset n(a) \geq n(b)$

The negation function that is used most often is $n(x) = 1 - x$. We will also use this function in the remainder of the thesis.

Intersection The intersection of two fuzzy sets μ_1, μ_2 is a fuzzy set μ, i. e., a function mapping each object x to a value in $[0, 1]$ representing the grade of membership of x in both sets μ_1, μ_2. Such a function is called a t-norm and defined as follows:

Definition 2.17 (T-norm)**.** *A function* $\top : [0, 1]^2 \to [0, 1]$ *is called t-norm, if*

- $\top(a, 1) = a$ *(neutral element),*
- $a \leq b \supset \top(a, c) \leq \top(b, c)$ *(monotonicity),*
- $\top(a, b) = \top(b, a)$ *(commutativity), and*
- $\top(a, \top(b, c)) = \top(\top(a, b), c)$ *(associativity).*

Thus, the intersection of two fuzzy sets μ_1 and μ_2 is defined elementwise as

$$(\mu_1 \wedge \mu_2)(x) = \top(\mu_1(x), \mu_2(x))$$

given some t-norm $\top$. Several t-norms exist. An example of a t-norm is $\top(a, b) = \min(a, b)$. All t-norms conform to the standard boolean conjunction if applied to the values 0 and 1 that stand for "false" and "true" respectively. For example, $\top(a, 0) = 0$ for all value of a.

We define a special form of t-norm, which we call continuous t-norm:

Definition 2.18 (Continuous T-norm)**.** *A t-norm* $\top$ *is called continuous if*

$$a < b \wedge c > 0 \supset \top(a, c) < \top(b, c)$$

An example for a continuous t-norm is $\top(a, b) = a * b$. The rational behind this definition is that the value $\top(a, b)$ of a continuous t-norm always depends on both arguments a and b, as opposed to, e. g., $\min(a, b)$ where the higher of the two values a, b has no influence.

Union The union of two fuzzy sets μ_1 and μ_2 is defined similarily to the intersection as

$$(\mu_1 \vee \mu_2)(x) = \bot(\mu_1(x), \mu_2(x)).$$

The function $\bot$ is some t-conorm:

Definition 2.19 (T-conorm)**.** *A function* $\bot : [0, 1]^2 \to [0, 1]$ *is called t-conorm, if*

- $\bot(a, 0) = a$ *(neutral element),*
- $a \leq b \supset \bot(a, c) \leq \bot(b, c)$ *(monotonicity),*
- $\bot(a, b) = \bot(b, a)$ *(commutativity), and*
- $\bot(a, \bot(b, c)) = \bot(\bot(a, b), c)$ *(associativity).*

Each t-norm $\top$ defines a dual t-conorm $\bot$ in the following way:

$$\bot(a, b) = 1 - \top(1 - a, 1 - b)$$

This formular reflects De Morgan's law: $a \vee b = \neg(\neg a \wedge \neg b)$.

Examples for t-conorms are $\perp(a,b) = \max(a,b)$, which is dual to $\top(a,b) = \min(a,b)$, and $\perp(a,b) = a + b - a * b$, which is dual to $\top(a,b) = a * b$.

Using the above definitions of complement, intersection and union of fuzzy sets together, we can interpret a propositional formula as a fuzzy set and provide a fuzzy evaluation of a propositional formula. Thus, the degree of truth of the sentence "the state s is near to a goal state" can be computed by a function $f(s)$ where f is a propositional formula representing the goal of the game. For example, in Tic-Tac-Toe, one subgoal for the `xplayer` is to complete one of the eight possible lines, that is

$$\begin{aligned} f \;=\; & \texttt{true}(\texttt{cell}(\texttt{a},1,\texttt{x})) \wedge \texttt{true}(\texttt{cell}(\texttt{b},1,\texttt{x})) \wedge \texttt{true}(\texttt{cell}(\texttt{c},1,\texttt{x})) \vee \\ & \texttt{true}(\texttt{cell}(\texttt{a},2,\texttt{x})) \wedge \texttt{true}(\texttt{cell}(\texttt{b},2,\texttt{x})) \wedge \texttt{true}(\texttt{cell}(\texttt{c},2,\texttt{x})) \vee \\ & \ldots \\ & \texttt{true}(\texttt{cell}(\texttt{a},3,\texttt{x})) \wedge \texttt{true}(\texttt{cell}(\texttt{b},2,\texttt{x})) \wedge \texttt{true}(\texttt{cell}(\texttt{c},1,\texttt{x})) \end{aligned}$$

We can interpret f as a fuzzy set $f(s)$ if we provide a suitable interpretation of the atoms $\texttt{true}(\ldots)$ of this formula as fuzzy sets $\texttt{true}(\ldots)(s)$, that is, if we provide functions $\texttt{true}(\texttt{t})(s)$ for all fluents t that estimate the likelyhood of making t true in the future if we are currently in state s. The method that we use to provide such functions and our choice of t-norm and t-conorm is presented in Chapter 5.

2.5. Answer Set Programs

In Chapter 7, we will use the Answer Set Programming (ASP) paradigm to prove properties of games. Answer sets provide models of logic programs with negation according to the following definition (for details, see, e. g., [Gel08]):

Definition 2.20 (Answer Set)**.** *Let P be a logic program with negation over a given signature, and let* ground(P) *be the set of all ground (i. e., variable-free) instances of rules in P. For a set M of ground atoms (i. e., predicates with variable-free arguments), the* reduct of ground(P) wrt. M *is obtained by deleting*

1. *all rules with some $\neg p$ in the body such that $p \in M$, and*
2. *all negated atoms in the bodies of the remaining rules.*

Then M is an answer set *for P if M is the least Herbrand model of the reduct of* ground(P) *wrt. M.*

According to this definition, an answer set is a least model of a logic program. Thus, it fulfils the closed world assumption, i. e., the assumption that only those things hold that are stated explicitely in the rules, while everything that is not explicitely stated is false. The same assumption applies to the GDL rules defining a game. For example, the rules of Tic-Tac-Toe (Figure 2.1) do not say anything about another legal action except `mark` and `noop`, Thus, according to the closed world assumption, we assume that only those two actions can be legal.

Answer sets provide an alternative semantics for GDL specifications. Unless additional rules are added, there is exactly one answer set for a GDL specification. This answer set coincides with the standard model for logic programs [ABW87].

In order to prove properties of a game, we need two additional constructs of answer set programs, which were introduced in [NSS99]:

weight atom A weight atom is a construct of the form

$$m \ \{ \ p : d(\vec{x}) \ \} \ n$$

The weight atom holds in an answer set M of a logic program iff the atom p has at least m and at most n different instances in M such that the instances satisfy $d(\vec{x})$. The numbers m or n can be omitted, in which case there is no lower or upper bound, respectively. A weight atom may occur in the same place as any other atom, that is, as head of a rule or as a literal in the body of a rule.

constraint A constraint is a rule with an empty head, that is, a rule of the form

$$\texttt{:-}\ b_1, \ldots, b_k$$

If a logic program P contains the constraint `:-` $b_1, \ldots, b_k$, answer sets that satisfy $b_1, \ldots, b_k$ are excluded. That means, all answer sets of P entail that at least one of $b_1, \ldots, b_k$ is false.

As an example, consider the following simple program:

```
1 init(cell(a,1,blank)). ... init(cell(c,3,blank)).
2 init(control(xplayer)).
3
4 cdom(xplayer).
5 cdom(oplayer).
6 t0 :- 1 { init(control(X)) : cdom(X) } 1.
7 :- t0.
```

The program contains the rules describing the initial state of Tic-Tac-Toe from Figure 2.1. In addition, there are two rules defining the predicate `cdom`, which encodes the domain of the `control` fluent. The rule in line 6 defines the predicate `t0` with the help of a weight atom. The rule says that `t0` holds if there is exactly one instance of `init(control(X))` that satisfies `cdom(X)`, that means, there is exactly one player who has control in the initial state, either `xplayer` or `oplayer`. Finally, the constraint in line 7 excludes any answer set that fulfils `t0`. Since `t0` is clearly entailed by the definition of the initial state, the above program has no answer set.

We use programs such as the one above, to prove properties of games in Chapter 7.

2.6. Summary

In this chapter, we defined what a game is and how its rules are described with the Game Description Language (GDL). We presented a summary of own

work [ST09b] on the semantics of GDL. Furthermore, we introduced Fuzzy Logic as a many-valued logic, which we will use in Chapter 5 for evaluating non-terminal states of the game. Finally, we introduced the Answer Set Programming (ASP) paradigm, which provides an alternative semantics for logic programs and, thus, game descriptions in GDL. We will use ASP in Chapter 7 to prove or refute hypotheses about properties of a game.

3. Components of Fluxplayer

In this chapter, we describe the structure and components of a general game player. In particular, we present how our player, Fluxplayer, works. In the following chapters we will present improvements to some of these basic components.

3.1. Overview

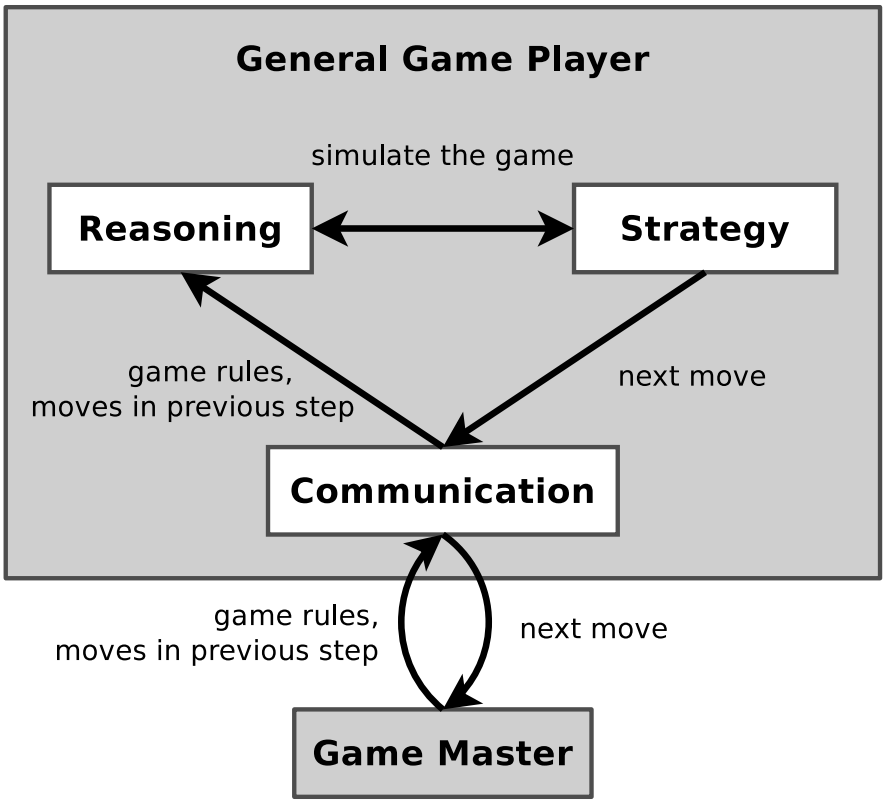

Figure 3.1.: The structure of a typical general game player and its communication with the game master.

A typical game playing system, as depicted in Figure 3.1, consists of three main components:

Communication The player has to communicate with a game master program to be informed about the progress in the match.

Reasoning The player also has to be able to reason about the game rules in order to know the current state of the match and the legal move options it has.

Strategy Finally, the player has to take a decision about which move to make next. This decision determines its strategy. Typically, choosing a move involves simulating the game to some extend in order to see the implications of choosing certain moves.

We will describe all three components in the following sections.

3.2. Communication

In the annual general game playing competitions players are set up to play matches of previously unknown games against each other. The infrastructure for the competitions requires a game master, i. e., a program that

- sends out the game descriptions to the players,
- collects the moves from the player and makes sure that moves are submitted on time,
- checks the legality of the submitted moves,
- informs all players about the previous moves of the competitors,
- informs all players about the end of the match.

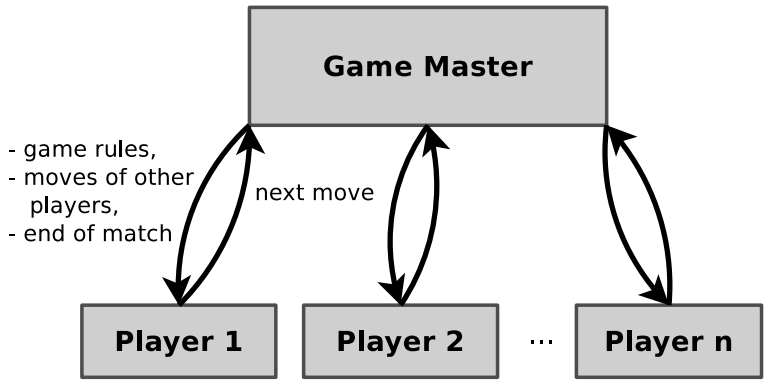

Figure 3.2.: The game master communicates game rules and game state information to the players. The players reply to the messages with their next move.

Each player of a match has to communicate with the game master. Technically, the players are simple HTTP servers that handle requests from the game master. Instead of serving web pages, their replies are the moves they want to make in the current state of the game. The communication protocol allows three requests from the game master, *start*, *play* and *stop*. The communication between the game master and a player is depicted in Figure 3.3 and defined in the following.

Start Message The first message in a match that the game master sends to the players is a start message. It has the structure:

```
(START <matchid> <role> <game rules> <start clock> <play clock>)
```

The start message contains the following fields:

`<matchid>` is a unique identifier for the match that is beginning.
`<role>` is the name of the role that the player receiving the message is supposed to play in the match. A game has one role for each player, e. g., white and black in chess.
`<game rules>` contains a formal encoding of the rules of the game. The language that is used is the Game Description Language that we introduced in Section 2.2.
`<start clock>` is the time in seconds that the player has to its disposal in order to learn or analyse the game before the match actually starts.
`<play clock>` is the time in seconds that the player has in every step of the game to decide which move to make.

After at most `<start clock>` seconds the players are supposed to reply to the start message with `READY` to signal that they are ready to play the match. If all players responded to the start message or the `<start clock>` is over, the game master sends the first play message.

Play Message The play message has the following format:

```
(PLAY <matchid> <moves>)
```

`<matchid>` is the identifier for the match to which the play message belongs.
`<moves>` contains the moves that all players selected in the previous step. For the first play message in a match, `<moves>` is `NIL`, that is, empty.

Based on the previous game state and the moves of all players every player can compute the current game state. The players have to respond to the play message within `<play clock>` seconds with their next move. The play clock was sent with the start message before and is the same for every step of the match. When all players have responded, the game master checks the legality of the moves and sends the next message. If the match is not over yet, this is again a play message. If some player sends an illegal move or does not respond in time, the game master selects an arbitrary legal move on that player's behalf. Additionally, in competitions, this player will typically receive zero points for the match, regardless of the outcome.

Stop Message If the match is over, that is, a terminal state of the game is reached, a stop message is sent instead of a play message:

```
(STOP <matchid> <moves>)
```

The stop message contains the same information as a play message and allows the players to compute the result of the game. It is considered polite if players respond to the stop message with `DONE`. However, since the game is officially over, this is not necessary.

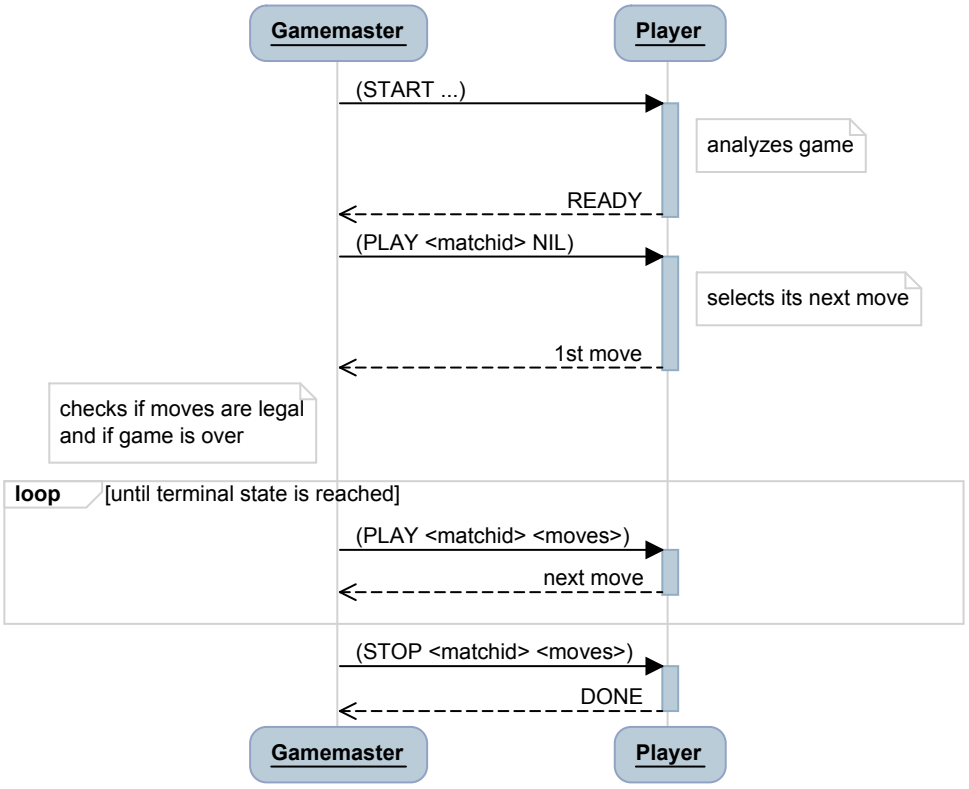

Figure 3.3.: Sequence of messages sent between the game master and a player during a match.

The communication component can be easily implemented in any programming language that supports socket communication. For Fluxplayer, we use a slight adaptation of the freely available NanoHTTPD [Elo10], which is implemented in Java. The only change that was necessary was to "serve" moves instead of web pages.

3.3. Reasoning

In order to play a game, the player has to reason about the rules of the game.

3.3.1. Objective of Reasoning

In order to submit legal moves, each player has to be able to compute the state the match is in and its own legal moves in the current state. However, in order to play good moves as opposed to just random legal moves, all players perform some kind of search of the game tree. That means, they simulate the game to some extend to see what happens under the assumption that certain decisions are made by each player. Therefore, the following reasoning tasks are performed by each player:

1. computing the initial state of the game;
2. given a state of the game, computing the legal moves of each player;
3. given a state and a move for each player, computing the successor state;
4. deciding if a state is terminal, i. e., if the match ends at this state;
5. given a terminal state, computing the outcome for each player.

In fact, these five reasoning tasks correspond to computing the components s_0, l, u, T and g of the game (R, s_0, T, l, u, g) (Definition 2.1) for a game description according to the semantics of a game description (Definition 2.14):

1. $s_0 = \{p \in \Sigma : D \models \texttt{init}(p)\}$ (the initial state);
2. $l = \{(r, a, s) : D \cup s^{\texttt{true}} \models \texttt{legal}(r, a)\}$ (the legality relation);
3. $u(A, s) = \{f \in \Sigma : D \cup s^{\texttt{true}} \cup A^{\texttt{does}} \models \texttt{next}(f)\}$ (the update function);
4. $T = \{s \in \mathcal{S} : D \cup s^{\texttt{true}} \models \texttt{terminal}\}$ (the terminal states);
5. $g(r, s) = n$ iff $G \cup s^{\texttt{true}} \models \texttt{goal}(r, n)$ (the goal function).

3.3.2. Implementation

All of the reasoning tasks presented in the previous section can be performed by computing the consequences of a logic program. Depending on the reasoning task, this program consists of the game description D possibly combined with a suitable encoding of the current state and the joint action of the players. For example, to compute the legal moves of a role r in a state s of a game described by the rules D, a player has to find all ground terms a such that $D \cup s^{\texttt{true}} \models \texttt{legal}(r, a)$, where $s^{\texttt{true}}$ is a suitable encoding of s as a logic program.

There are two main ways to compute the consequences of a logic program: *bottom-up* and *top-down* reasoning.

Bottom-up reasoning The bottom-up approach computes a complete model for a program. Once a model is computed, the answers to the reasoning task can just be read from the model.

Bottom-up reasoning can, for instance, be implemented using an answer set programming system as described in Section 2.5. However, current ASP systems (e. g., clingo [oP10], DLV [LPFe10], smodels [Sim08]), come with a considerable overhead: They need to ground the input program, that is, replace each rule

by every ground instance of it. Grounding game descriptions is expensive with respect to both time and memory consumption. In the worst case, a grounded game description is exponentially larger than the one with variables. For some games, grounding the game description is not feasible because of memory or time consumption. This makes ASP systems an unsuitable solution for the reasoning tasks mentioned above because a general game player should be able to play as many games as possible.

Top-down reasoning Instead of computing a complete model, the top-down approach only checks whether some formula is entailed by the program. Usually, that formula contains existentially quantified variables and an answer of the reasoning system is a substitution for these variables such that the formula is entailed. One top-down reasoning approach is resolution. We use a variant of resolution, first-order SLDNF resolution [Llo87], which is implemented in every prolog system. As the name suggests, first-order SLDNF resolution uses first-order logic. Thus, grounding is not necessary. Standard SLDNF resolution can be used to perform all reasoning tasks mentioned above.

Translation from GDL to Prolog For efficiency, we directly use the prolog inference by automatically translating the game rules D into prolog rules at the start of the match. Note that for some of the reasoning tasks, we need to add additional rules containing a suitable encoding of the current state and the moves chosen by the players. For example, to compute the legal moves of a role r in a state s, we have compute all instances of a such that $D \cup s^{\texttt{true}} \models \texttt{legal}(r, a)$. In this case, the rules $s^{\texttt{true}}$ that encode the state s as a logic program need to be added to the game description D.

Because we want to avoid asserting and retracting facts to and from the prolog rule database, we encode states of the game by prolog terms and add an additional argument to all predicates that depend on the current state. These predicates are, for example, `legal`, `goal`, and `terminal`, but may include game specific predicates. The same is done for the previous moves of the players and predicates that depend on these moves, e. g., `next`. For example, the rule

```
1 legal(P,mark(X,Y)) :-
2     true(control(P)), true(cell(X,Y,blank)).
```

from Figure 2.1, which describes when a player `P` can do the move `mark(X,Y)`, is translated to the following prolog rule:

```
1 legal(P, mark(X,Y), State) :-
2     true(control(P), State),
3     true(cell(X,Y,blank), State).
```

We translate a game description D from GDL to a prolog program P according to the following schema:

1. Compute the dependency graph of the game rules (cf. Definition 2.6).

2. Replace every n-ary predicate $p(\vec{t})$ by $p(\vec{t}, Moves, State)$, if `p` is connected to `does` and `true` in the dependency graph.
3. Replace every n-ary predicate $p(\vec{t})$ by $p(\vec{t}, Moves)$, if `p` is connected to `does`, but not to `true` in the dependency graph. Note that this includes `does` itself.
4. Replace every n-ary predicate $p(\vec{t})$ by $p(\vec{t}, State)$, if `p` is connected to `true`, but not to `does` in the dependency graph. Note that this includes `true` itself.
5. Add the following additional rules:

```
1 true(Fluent, State) :- member(Fluent, State).
2 does(R, M, Moves) :- member(does(R, M), Moves).
3 member(X,[X|_]).
4 member(X,[_|L]) :- member(X,L).
```

With this representation of a game description as prolog rules, reasoning about a state s and joint move A of a game can now be done without adding the sets of rules $s^{\texttt{true}}$ and $A^{\texttt{does}}$. Instead, we use suitable encodings of state s and joint action A as ground terms and use these terms in the query posed to the prolog system. For example, the legal moves of a role r in state s can be computed by finding all substitutions for the variable A in the query $P \models (\exists)\texttt{legal}(r, A, s^{term})$, where P is the prolog program obtained from the game description and s^{term} is a prolog term representing the state s.

We use prolog lists to represent states and joint actions:

Definition 3.1 (Prolog State Representation)**.** *Let $s = \{f_1, f_2, \ldots, f_n\}$ be a state in a game, then $s^{term} = [f_1, f_2, \ldots, f_n]$ is a prolog representation of s.*

Definition 3.2 (Prolog Joint Action Representation)**.**
Let $A = \{r_1 \mapsto a_1, r_2 \mapsto a_2, \ldots, r_n \mapsto a_n\}$ be a joint action in a game, then $A^{term} = [does(r_1, a_1), does(r_2, a_2), \ldots, does(r_n, a_n)]$ is a prolog representation of A.

After translating a game description D to the prolog program P, we can solve our five reasoning tasks using the previous definitions as follows:

1. The initial state s_0 of a game is the set of all fluents f such that $P \models \texttt{init}(f)$.
2. The legal moves of role r in a state s is the set of all actions a such that $P \models \texttt{legal}(r, a, s^{term})$.
3. The state that results from executing joint action A in state s is the set of all fluents f such that $P \models \texttt{next}(f, A^{term}, s^{term})$.
4. A state s is terminal if and only if $P \models \texttt{terminal}(s^{term})$.
5. The goal value of a role r in a state s is n ($g(r, s) = n$) if and only if $P \models \texttt{goal}(r, n, s^{term})$.

3.4. Strategy

The strategy of a player is responsible for selecting the moves that the player performs from the legal moves. Therefore, the strategy is the component that has the biggest influence on whether the player wins the game or not. All current general game playing systems use some form of search to evaluate moves and then select the best move according to that evaluation. We will discuss different search algorithms in Chapter 4 and the heuristics used by Fluxplayer to guide the search in Chapter 5.

3.5. Summary

In this chapter we presented the structure of general game players in general and Fluxplayer in particular. We described how players communicate with the game master program and how we reason about game descriptions, that is, how we compute legal moves, successor states, etc. from a game description.

4. Game Tree Search

Search is a fundamental technique that is used in every general game player today. All players search the game tree in order to determine which moves are advantageous. Intuitively, players reason about what would happen if certain moves were taken in some situation and compare the resulting state to the other options.

In this chapter, we will first elaborate on what we mean by searching the game tree. Then, we present the two search methods that are used for General Game Playing today. Finally, we will describe the search algorithm that we use in Fluxplayer.

4.1. Game Tree Search In General

The game tree is a tree composed of the states of the game as nodes where the initial state of the game is the root of the tree and terminal states constitute the leaf nodes. The edges of the tree are the joint actions of the players that lead from one state to another. For example, Figure 4.1 shows a partial game tree of Tic-Tac-Toe.

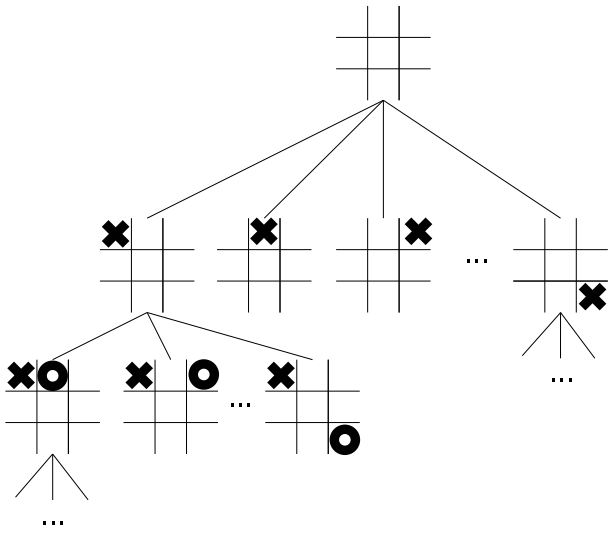

Figure 4.1.: Partial game tree of Tic-Tac-Toe.

Players search the game tree in order to determine the best action for themselves in the current state. Therefore, the player assigns values to the states (i. e.,

nodes) or the actions (i. e., edges) in the tree such that higher values coincide with states or actions that are more advantageous for the player. In the following we will only discuss values for states. The value of an action can be seen as the value of the state that is reached by execution the action.

The best action in the current state is the action that leads to the successor state with the highest value. In a terminal state, we can immediately determine the value for each player from the goal rules of the game description. However, for non-terminal states or actions, no evaluation is defined by the game rules. Therefore, the value of an intermediate state s must be computed from the values of the terminal states that can eventually be reached from s. How this computation is implemented depends on the search algorithm, the type of game, and what is assumed about the strategy of the opponent players.

Computational Complexity The main problem with game tree search is that for all but very simple games the game tree is too large to be searched completely under the given time constraints. For example, solving the game of Checkers by brute-force search of the game tree took over 15 years on an average of 7 processors with 1.5 to 4 GB of RAM and produced an endgame database with $4 * 10^{13}$ entries [SBB+07]. Checkers has approximately $5 * 10^{20}$ reachable states [SBB+07]. Other games are even larger: The number of legal chess positions is estimated to be 10^{43} [Sha50]. On a standard 19x19 Go board there are $2 * 10^{170}$ reachable positions [TF06].

Because game trees are so large, we can only search a small part of the game tree in general. Thus, we cannot compute the exact value of an intermediate state, i. e., the value a player would receive if the game was played from this state on and every player played optimally for itself. Instead, players must estimate the value based on the part of the game tree that they can search within the given time. We call the expanded part of the game tree search tree. The root of the search tree is the current state of the game.

Heuristic Search vs. Monte-Carlo Tree Search There are two fundamentally different ways of selecting the part of the game tree to search: heuristic search and Monte-Carlo tree search (MCTS). Figure 4.2 depicts both strategies. Intuitively, heuristic search can be seen as looking a number of steps ahead to see how the game evolves. On the other hand, MCTS expands only a few possible branches of the game, but simulates them until a terminal state is reached.

Players using MCTS can be considered knowledge-free. These players see a game as a state machine, without considering any structure in the game state itself. That means, a game state is considered as one atomic item. Almost all of the successful GGP system today use an essentially knowledge-free approach, namely Monte-Carlo tree search or one of its extensions.

On the other hand, heuristic search can be seen as a knowledge-based approach. The heuristics exploits knowledge about the internal structure of the game in order to evaluate states. Since our focus is on knowledge-based general game

playing, Fluxplayer uses a heuristic search method that we will describe in Section 4.4.

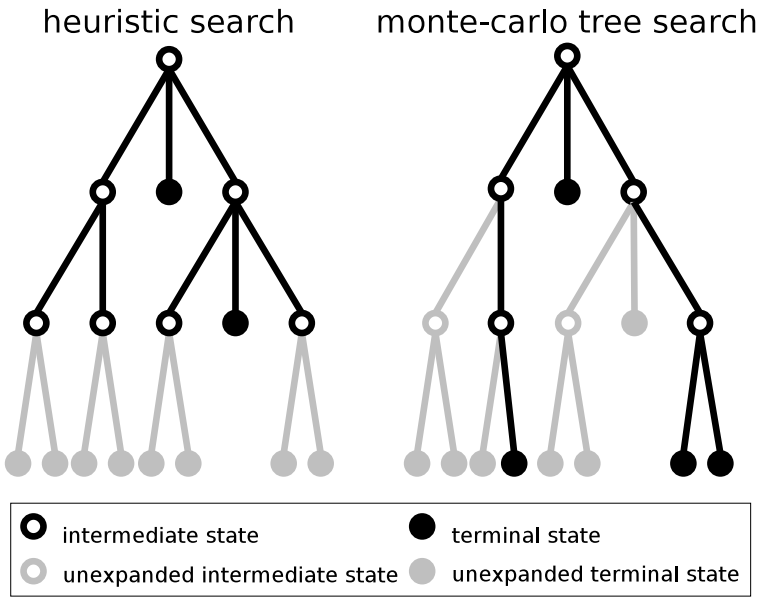

Figure 4.2.: Expanded parts of the game tree for heuristic search and Monte-Carlo tree search.

We will discuss heuristic search and MCTS in more detail in Section 4.2 and Section 4.3, respectively.

4.2. Heuristic Search

Heuristic search expands all branches of the game tree but typically stops at a certain depth. Thus, the search tree of heuristic search may have non-terminal states of the game as leaf nodes. A heuristic evaluation function is used to estimate the value of the leaf nodes because there is no evaluation defined for non-terminal states.

Definition 4.1 (Heuristic Evaluation Function (Heuristics))**.** *Let* (R, s_0, T, l, u, g) *be a game with states* $\mathcal{S}$*. A function* $h : R \times \mathcal{S} \to \mathbb{R}$*, assigning a real value to every role and game state is called* heuristic evaluation function *iff*

$$(\forall r \in R, s \in \mathcal{S}) \;\; 0 \leq h(r, s) \leq 100$$

We define the heuristic values to be between 0 and 100 because they should estimate the goal value a player can achieve and all goal values are in the interval $[0, 100]$ (see Section 2.2). Of course, other definitions are possible.

In traditional computer game playing, heuristics are typically designed by domain experts and contain features that are specific to the game that is played. For

example, the heuristics for a chess program may contain features such as material value (each chess piece is assigned a certain value), bonuses and penalties for certain positions (e. g., knights are stronger in the centre), and penalties for attacked and undefended pieces. These features are specific to Chess and have no meaning or relevance in other games. Hence, they are not suitable to use for general game playing. In General Game Playing, we do not know the game in advance. Hence, no fixed game specific heuristics can be used. Instead, a general game player using heuristic search must construct its own heuristics based on automatic analysis of the game before starting to search the game tree. We will discuss how to construct such a heuristics in Chapter 5.

Given a heuristics, we can compute an estimate for the value of each non-terminal state of the game. Values for terminal states are defined by the game rules. Leaf nodes in the search tree, i. e., the part of the game tree expanded by the search, can be non-terminal or terminal states of the game. Thus, we can define the value of a leaf node in the search tree as follows:

Definition 4.2 (Value of Leaf Nodes)**.** *Let (R, s_0, T, l, u, g) be a game with states $\mathcal{S}$ and h be a heuristic evaluation function for the game. For every role $r \in R$ and leaf node $s \in \mathcal{S}$ of a (partially expanded) game tree*

$$v(r,s) = \begin{cases} g(r,s), \textit{ if } s \in T \\ h(r,s), \textit{ otherwise} \end{cases}$$

Values of non-leaf nodes are determined by the values of their successors states, which are either leaf nodes or non-leaf nodes in the search tree, themselves. Different algorithms exist to compute the values of non-leaf nodes depending on the type of game. For example, in single-player games, the value of a non-leaf node is just the value of the best successor state because the (only) player of the game has complete control of the game and can choose the best action. This is not the case for multi-player games because the successor state may depend on the actions of the opponents, which a player has no influence on.

We will shortly discuss heuristic search for three classes of games in the following subsections.

4.2.1. Turn-taking N-Player Games

Turn-taking games (sometimes called sequential games) are games in which players take turns in making their moves. Note that in our game model (see Definition 2.1 on page 5) every player has to make a move in every step. Thus, we consider a game turn-taking if in every state just one of the players has more than one legal move. The other players are only allowed to execute one action, typically called `noop`. By this definition, single-player games are considered turn-taking, too.

The value of a state of such a game can be computed with the max^n algorithm presented in [LI86]. The rational of the max^n algorithm is that every player tries to maximise its own reward. Thus, the value $v(r, s)$ of a non-terminal

state s for role r is the value of the successor state $u(A', s)$ that has the highest value for the role r_t who's turn it is:

$$\begin{aligned} v(r,s) &= v(r, u(A', s)) && (4.1)\\ \text{with} \quad & (\forall r_i \neq r_t)\ (\forall a_1, a_2) l(r_i, a_1, s) \wedge l(r_i, a_2, s) \supset a_1 = a_2 && (4.2)\\ \text{and} \quad & v(r_t, u(A', s)) = \max_{A \mid (\forall r_i) l(r_i, A(r_i), s)} v(r_t, u(A, s)) && (4.3) \end{aligned}$$

Equation 4.2 defines r_t as the role of the game such that all other roles r_i have only one legal move in state s. Equation 4.3 defines A' as the legal joint action in state s such that the value of role r_t in the successor state $u(A', s)$ is maximal.

The values for all nodes in the search tree are computed recursively using Equation 4.1 for non-leaf nodes and Definition 4.2 for leaf nodes of the tree.

4.2.2. Turn-taking, Constant-sum, Two-player Games

This class of games is a special case of the one above. A game is constant-sum if the rewards for all players add up to the same (constant) sum in every terminal state. Constant-sum games are often termed zero-sum games, although in our setting the sum of rewards will not be zero but usually 100 (winning means 100 and losing means 0 points). Every constant-sum n-player game with sum s can be easily transformed into a zero-sum game by subtracting $\frac{s}{n}$ from the goal value of every player.

Effectively, a constant-sum two-player game is perfectly competitive: Every advantage for one player is a disadvantage for the other one. The class of turn-taking, constant-sum, two-player games contains many popular board games, such as Chess, Checkers and Go.

The value of a state of such a game can be computed with the minimax algorithm [RN95]. The assumption of minimax is that the first player tries to maximise his own score while the second player tries to minimise the score of the first player. Because the game is a constant-sum game, a player who minimises its opponent's score automatically maximises its own score.

The minimax algorithm is a slight variant of max^n where only the value of the first player needs to be computed. There are several improvements to the minimax algorithm, such as alpha-beta pruning [RN95] or Negascout [Rei89]. Both exploit the fact that some nodes of the game tree do not have to be expanded because the actions would not be selected by a rational player.

Alpha-beta pruning Alpha-beta pruning is a method to keep track of the lower bounds α and upper bound β of the value of a state during search. The lower bound α is the value that the first player can at least win if he plays the best of his moves that were searched so far. The upper bound β is the value that the first player will at most win if the opponent plays his best move. Remember, that minimax and alpha-beta pruning are only applicable to constant-sum two-player games, i. e., games where every gain of one player is a loss for the other player.

Using these two bounds, the search can stop evaluating an alternative move when it is clear that the move will not be better than the previously found best move. There are two possible reasons for cutting of the search:

Alpha cut-off Suppose, in some state s, the second player has a move that leads to a goal value smaller than α, i. e., the second player can enforce that the first player will get a reward less than α in this state. Then, the state s need not be explored further, because the first player would not select a move leading to s. After all, he has some other option that gives him a reward of at least α.

Beta cut-off Conversely, suppose in some state s (other than the initial one) the first player has a move that leads to a goal value greater than β. Then the second player would not select a move leading to s because he has a different option that guarantees a value of at most β for the first-player and, thus, a higher value for the second player.

4.2.3. General N-player Games

Both the max^n and the minimax algorithm rely on the fact that only one player can choose an action in each state. However, in GGP, games are simultaneous in general, that is, every player selects a move in every state. In this case, the max^n algorithm does not apply because it is unclear whose value should be maximised in a state.

In this general case of games, the value of a state can be computed using the concept of a Nash equilibrium [LB08]. A Nash equilibrium is a tuple of strategies, one for each player, such that none of the players can improve his reward by solely changing his strategy. For turn-taking games, the max^n algorithm computes the value of the Nash equilibrium, i. e., the rewards the players would get if they played according to their respective strategy in the equilibrium. While interesting from a theoretical point of view, actually using Nash equilibria to compute the value of a state poses several problems:

- Computing Nash equilibria is expensive in the general case. For example, [DGP06] showed that computing Nash equilibria for 4-player games is PPAD-complete. PPAD is a complexity class which is considered intractable.
- John Forbes Nash showed in 1950 that there exists a Nash equilibrium in every finite game [Nas50]. However, there may be more than one equilibrium and the equilibria might yield different rewards.

Because of these technical difficulties, general n-player games (including those with simultaneous moves) are often treated as sequential games by making the assumption that players move sequentially. In order to do this, artificial nodes are added to the game tree as depicted in Figure 4.3. These nodes are artificial in the sense that they do not correspond to an actual game state, but to a game state in which some of the players have already chosen a move but others have not.

For example, the state $s1$ in Figure 4.3 is the result of the joint action $\{r_1 \mapsto a, r_2 \mapsto c\}$ executed in state $s0$ ($s1 = u(\{r_1 \mapsto a, r_2 \mapsto c\}, s0)$), where r_1 and r_2 are the roles of the game. In the sequentialised game tree, the artificial node $n1$ refers to the situation where player r_1 decided to make move a but player r_2 has not decided which of the moves, c or d, to make. Therefore, $n1$ has no associated state of the game. However, it can be represented by the pair $(s0, \{r_1 \mapsto a\})$ of the state $s0$ and the partial joint move $\{r_1 \mapsto a\}$.

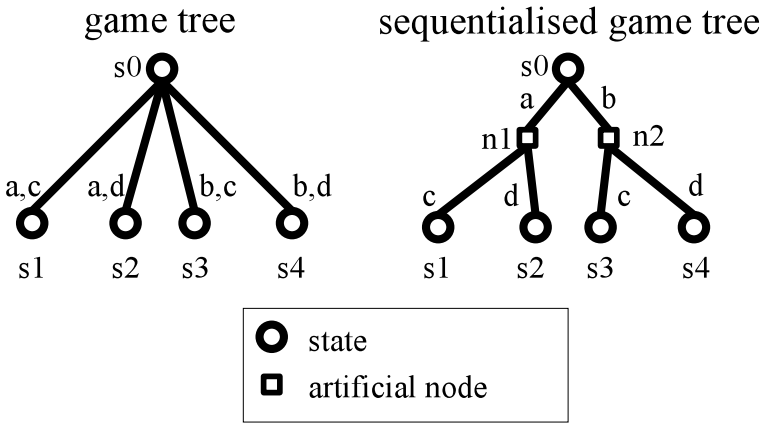

Figure 4.3.: Treating a general game as sequential by adding artificial nodes. In this two-player game the first player chooses action a or b simultaneously to the second player choosing action c or d. In the sequentialised version of the game the second player chooses his move after the first player.

After this transformation standard max^n search can be used to compute the value of each node, albeit with an error: All players except the first, are overestimated, that is, they are modelled stronger than they are in reality. max^n will select their best response to the first player's move although in the actual game the players will not get to know the first player's move until they have decided their own move. Thus, the value of a state computed by max^n for the first player of the game is typically lower than the real value of the state. For example, consider the popular game of Rock-paper-scissors where two players simultaneously select either rock, paper, or scissors. Rock defeats scissors, scissors defeats paper, and paper defeats rock. Thus, all three actions have an equal chance of winning the game if the move of the opponent is unknown. However, in the sequentialised game, the player who moves second can always win the game by selecting a perfect response to the move of the first player. Thus, the value of every move of the first player will be computed as 0 by the max^n algorithm, although the real value of each move is 50 (if 0 means loosing, 100 means winning, and 50 stands for a draw).

4.3. Monte-Carlo Tree Search (MCTS)

In contrast to heuristic search, MCTS expands only some branches of the tree but follows these until a terminal state is reached. This is possible because all branches in the game tree are finite (see Section 2.1.3 on page 7). The value of an intermediate state s is then typically computed as the average reward of the terminal states reached by the expanded paths that start in s. Thus, no state evaluation function is necessary. This makes the implementation easier than that of a player using heuristic search. All winners of the International GGP competitions since 2007 use Monte-Carlo tree search [FB08, MC11]. However, the recent addition of heuristics to guide Monte-Carlo simulations [SKG08, FB10, KSS11] suggests that adding knowledge acquisition techniques, such as learning, improves the performance of a MCTS player further.

4.4. Search Method of Fluxplayer

The search method used by our player is an iterative-deepening depth-first search with a heuristics for evaluating non-terminal states. We treat every game as sequential, as described in Section 4.2.3. Thus, we can apply max^n search to all games. For the special class of two-player constant-sum games we use the minimax search with alpha-beta pruning as described in Section 4.2.2. In Chapter 7, we describe a method for proving if a game is a zero-sum game.

Furthermore, we use the following well-known enhancements of max^n for all classes of games:

- Values for visited states are stored in a transposition table [Sch89], i. e., a cache. Thus, when the same state is reached on a different path in the game tree it need not be expanded or evaluated again.
- History heuristics [Sch89] is used for determining the order in which the successor states of a state are expanded. The order is based on the values of the successor states in the previous iteration of the iterative deepening search. This enhancement leads to more cut-offs of the alpha-beta pruning and, thus, speeds up search.

One problem of iterative-deepening depth-first search is, that the maximal depth of the search is low in games with a large branching factor, that is, games where players have a large number of possible moves. The reason for this is that the number of nodes in the game tree up to a certain depth depends greatly on the branching factor of the game. Thus, games with a high branching factor can only be search to a very small depth. This limits the accuracy of the computed state values. To mitigate this problem, we do not use a uniform depth limit for the whole game tree. Instead, states that seem to be advantageous for the player that is on control are searched deeper than their siblings that are less promising. We judge if a state is advantageous for role r by observing the value $v(r, s)$ that this state was assigned in the previous iteration of the iterative deepening search.

Let us briefly recapitulate how iterative deepening search works, before we explain our non-uniform depth search. Iterative deepening search iteratively executes a depth limited search from the current state of the game and increases the depth limit $\hat{d}$ (usually by 1) for every iteration. Depth limited search searches the game tree up to the current depth limit $\hat{d}$. The depth limited search can be thought of as a recursive algorithm that keeps track of a depth limit $d(s)$ for each state s, as depicted in Algorithm 5. This algorithm is called with $DLSearch(s_0, \hat{d})$, where s_0 is the current state of the game. If the depth limit for a state is smaller than 1, the state is not expanded but evaluated according to the state evaluation function (line 3). Otherwise the successor states are searched with the depth limit reduced by one (line 9) and the value of the state is computed from the value of the successor states using max^n (line 11).

Algorithm 1 $DLSearch(s, d)$ – Depth limited search of state s to depth d.

Input:
s, a game state
d, the depth limit for the search

```
1:  if d < 1 then
2:      for all r do
3:          v(r, s) := h(r, s)
4:      end for
5:  else
6:      Let s_1, ..., s_n be the successor states of s.
7:      for all i ∈ {1, ..., n} do
8:          d(s_i) := d − 1
9:          DLSearch(s_i, d(s_i))
10:     end for
11:     Compute v(r, s) for all r using the max^n algorithm (cf. Section 4.2.1).
12: end if
```

To change $DLSearch$ to a depth limited search with non-uniform depth, we do the following.

Order the Successor States We want to have higher depth limits for those successor states that are more promising. Therefore, we order the successor states $s_1, \ldots, s_n$ by the values $v(r, s_1), \ldots, v(r, s_n)$ that they were assigned in the previous iteration of the iterative deepening search[1]. Hence, without loss of generality we assume that

$$v(r, s_1) \geq v(r, s_2) \geq \ldots \geq v(r, s_n)$$

in line 6 of Algorithm 1, where r is the role that is in control in state s.

Assign Depth Limits Depending on the Value of the State The depth limits of the successor states $s_1, \ldots, s_n$ of s shall be such that the most promising state is searched deepest. We use a simple linear function for the depth limits of the successor states, such that, the depth limit of the most promising state (i. e., s_1) is $d(s) - 1$ and the depth limit for the least

[1] An arbitrary order is used for the first iteration.

promising state (i. e., s_n) is $d(s)/2$. We achieve this by changing line 8 in Algorithm 1 to the following equation:

$$d(s_i) = (d(s) - 1) * \left(1 - \frac{i-1}{2 * (n-1)}\right) \tag{4.4}$$

Note, that with this equation even if all states have the same value, we will select arbitrary states to explore deeper than others. This ensures that the search focuses on a few paths if the current value of the states gives no indication on which of the states is actually better for the player who is in control.

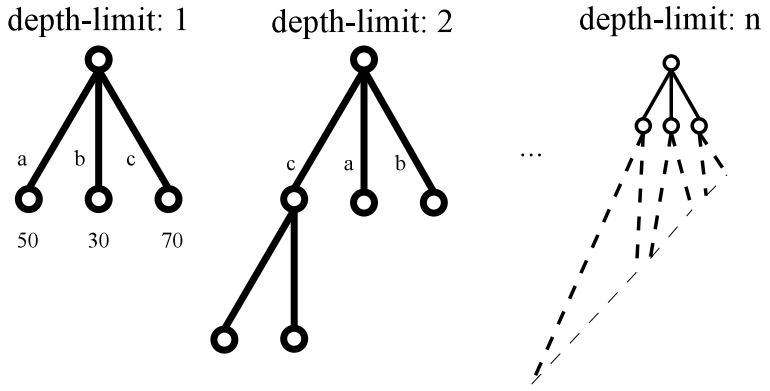

Figure 4.4.: Search trees for different iterations of iterative-deepening search with non-uniform depth-limit. States with a higher values in previous iterations will be explored deeper in the next iteration.

Figure 4.4 shows a sequence of search trees for iterative deepening search (IDS). IDS starts with a depth limit of 1, that is, the depth limit $d(s)$ of the root node of the tree is 1. Thus, the states in level 1 are not expanded but evaluated with the heuristic evaluation function. These values determine the order c, a, b for expanding the states in the next iteration. In the next iteration, the depth limit of the root node is 2. Thus, the depth limit of the first successor state (c) is 1 and this state is expanded further. The remaining states are not expanded in this iteration because their depth limits are smaller than 1 according to Equation 4.4. In the end, this procedure produces a search tree as depicted on the right-hand side of Figure 4.4: The most promising path is explored to depth d and the remaining paths are explored to increasingly lower depths.

4.5. Summary

In this chapter, we described game tree search because it is used in every general game playing system. Furthermore, we described the heuristic search method, that we use in Fluxplayer. Although Monte-Carlo tree search (MCTS) players were more successful in the last few years, we believe that further progress is only

possible with a knowledge-based approach. This belief is supported by the current effort to enhance MCTS based systems with knowledge aquisition techniques. The boundaries between knowledge-free and knowledge-based approaches are not fixed. While pure MCTS can be deemed knowledge-free, adding heuristics to guide Monte-Carlo simulations clearly adds some form of knowledge.

Both heuristic search methods and MCTS benefit from the addition of knowledge about the game. Therefore, the focus of the remainder of the thesis lies on automatic game analysis and knowledge aquisition techniques with the goal to automatically construct heuristics. Although we developed these techniques with the heuristic search method of Fluxplayer in mind, they can be used to enhance other search methods, such as MCTS, with knowledge about the game.

5. Generating State Evaluation Functions

When using game tree search to find good moves, we have to evaluate states in order to estimate the potential for our player to win the game with the available moves. In General Game Playing, we do not know in advance which game is played. Hence, the state evaluation function used for this purpose cannot be predefined for a specific game. Instead, it has to be automatically derived for the game at hand. As explained in the previous chapter, we want to use a heuristics to estimate the value of a state instead of random simulations of the game. In the remainder of this work the terms heuristics and state evaluation function are interchangeable.

One way of generating heuristics for a game is to learn them by perceiving the outcomes of real or simulated matches of the game. However, for learning evaluation functions, two problems have to be solved:

Defining the structure of the evaluation function A typical evaluation function is composed of features, often in the form of a linear combination. Each feature is a function that assigns a value to certain structures in the game state. For example, the number of white pieces on the board is a feature in Chess. If the evaluation function is a linear combination of features, each feature is given a weight that represents the influence or importance of the feature. These weights are the values that have to be learned or be defined by hand. In Chess, for example, the number of white pieces on the board should get a positive weight in the evaluation function for the white player, while the number of black pieces should get a negative weight. Features for specific games are typically hand selected by domain experts. Automatically finding features that are meaningful for a game is difficult and time consuming [Faw96, Gün08].

Learning the parameters Once the features for the heuristics are selected and the structure of the heuristics is fixed, the weights of the features and possibly other parameters of the evaluation function have to be learned. There is a multitude of learning algorithms to choose from. However, all of them have in common that for a high quality of the learned weights a large amount of training data is necessary. In our case, training data consists of the move history and the outcomes of real or simulated matches of the game. For previously unseen games no expert matches exist to learn from. While it is possible to simulate matches, simulations are time consuming. Furthermore, the quality of the simulations depends on the strategy that is used during the simulation. However, for previously unseen games we do not know a good strategy.

In GGP competitions, a player has typically only a few minutes to analyse a game. This time is too short to run expensive algorithms for finding features and learning their weights. Thus, we decided to use an approach that does not need learning. Instead, our heuristics is directly generated from the rules of the game, specifically the rules defining the terminal and the goal states of the game.

In Section 5.1, we will present our basic evaluation function and how it is obtained from the goal and terminal rules of the game. We improve this function by taking additional knowledge about the structures in the game into account. We explain how we find structures such as game boards and orders in the game in Section 5.2 and show how these structures can be used to improve the evaluation function in Section 5.3. In Section 5.4, we show the effectiveness of our heuristics with the help of real matches played in a GGP competition. The results from this chapter were published in [ST07b].

5.1. The Basic State Evaluation Function

The idea for our heuristics is to calculate the degree of truth of the formulas that define the predicates `goal` and `terminal` in the state to evaluate. Informally, that means we count how many of the conditions for winning the game are fulfilled in a state. The underlying assumption is that the goal of the game can be reached gradually and that game states are relatively stable, i. e., only few properties change from one state to the next one. This assumption is justified by the fact that games in GDL are described in a modular fashion: Game states are composed of fluents and there are rules defining when these fluents are contained or not contained in a state. Thus, it is natural to describe games in such a way that the actions of the players only influence a small part, i. e., some fluents of a state. For example, in a typical description of Chess, there is a fluent for each piece on the board denoting the position of this piece. A typical action, such as moving a piece, only changes the location of this one piece. Consequently, the remaining fluents of the state will not change. Thus, game states are relatively stable and most of the conditions for winning the game will persist from one state to the next.

We use fuzzy logic (introduced in Section 2.4) to compute the degree of truth of goal and terminal formulas with respect to a state s. In other words, we take a formula f that describes the situation where a player r wins the game, for example, $f = \texttt{goal}(r, 100)$. Then, we interpret this formula f as a fuzzy set containing a state s to a certain degree. This degree corresponds to an estimate of how near s is to a state in which f holds, that is, how near we are to winning the game. We call the degree to which state s is in the fuzzy set f the degree of truth of formula f with respect to state s. Although intuitively, the degree of truth of $\texttt{goal}(r, 100)$ with respect to state s is an evaluation function of the state s for role r, our actual evaluation function is a bit more complex.

We will proceed as follows to define our evaluation function:

- First, we define the fuzzy evaluation $eval(f, s)$ of arbitrary formulas f with respect to state s (Section 5.1.1).
- The definition of $eval(f, s)$ contains several parameters whose choices we will motivate in Sections 5.1.2 and 5.1.3.
- In Section 5.1.4, we will discuss some theoretical properties of $eval(f, s)$.
- Finally, our state evaluation function will be defined in Section 5.1.5 based on the definition of $eval(f, s)$.

5.1.1. Evaluation of Formulas

Before we present our the fuzzy evaluation of formulas, we need to define what we consider as a formula.

Definition 5.1 (GDL Formula)**.** *A GDL formula f of a game description D is a first order formula with the usual connectives for conjunction $\wedge$, disjunction $\vee$, negation $\neg$, and existential quantifiers $\exists$. The atoms of f are atoms over the signature (relation symbols and function symbols) of D.*

Let us now define our fuzzy formula evaluation $eval(f, s)$, that is, the function computing the degree of truth of a formula f with respect to state s:

Definition 5.2 (Fuzzy Formula Evaluation)**.** *Let D be a set of GDL rules and parameter p be a real value with $0.5 < p \leq 1$. Furthermore, let a denote GDL atoms, f and g denote arbitrary GDL formulas, and $\top$ denote an arbitrary t-norm with dual t-conorm $\bot$. We define a fuzzy evaluation function for GDL formulas wrt. a game state s as follows:*

conjunction *If f and g contain no common variables:*

$$eval(f \wedge g, s) = \top(eval(f, s), eval(g, s)) \tag{5.1}$$

Thus, conjunctions are evaluated by evaluation every conjunct and using the t-norm $\top$ to combine both values.

disjunction *If f and g contain no common variables:*

$$eval(f \vee g, s) = \bot(eval(f, s), eval(g, s)) \tag{5.2}$$

Thus, disjunctions are evaluated by evaluation every disjunct and using the t-conorm $\bot$ to combine both values.

negation

$$eval(\neg f, s) = 1 - eval(f, s) \tag{5.3}$$

Thus, we use the usual negation function from fuzzy logic, $n(x) = 1 - x$, to evaluate negated formulas.

atoms defined by rules *For every atom a except $\texttt{distinct}(t_1, t_2)$, $\texttt{true}(t)$, and $\texttt{does}(r, m)$, let $\texttt{a}_1 :- \texttt{b}_1$, ..., $\texttt{a}_n :- \texttt{b}_n$ be all rules in D such that a unifies with the head a_i of each rule with unifier σ_i, that is, $a_i\sigma_i = a$ for all $i \in 1 \ldots n$. In this case,*

$$eval(a, s) = eval(b_1\sigma_1 \vee b_2\sigma_2 \vee \ldots \vee b_n\sigma_n, s) \tag{5.4}$$

This rule applies to all atoms that are defined by rules in the game description. We replace such an atom a by a disjunction of the bodies of the rules whose head matches a. If there is no matching rule for a, then a is unsatisfiable. In this case $eval(a, s) = 0$.

other *For all GDL formulas f that do not match any of the rules above:*

$$eval(f, s) = \begin{cases} p & \text{if } D \cup s^{\texttt{true}} \models (\exists) f \\ 1 - p & \text{otherwise} \end{cases} \tag{5.5}$$

As defined in Section 2.3, $\models$ denotes entailment by the standard model of logic programs and $s^{\texttt{true}}$ is a representation of state s as logic program.

This rule defines the degree of truth of a formulas f to be the value p if f holds in s, that is, if the game description together with a suitable encoding of s as a logic program entails f. If the formula f does not hold in s, the value is $1 - p$. We will discuss how to choose a value for p in Section 5.1.2.

While Equations 5.1 to 5.4 define the recursive cases of the function *eval*, Equation 5.5 defines the base case. Observe that the formula f in Equation 5.5 can be either

- a basic atom of the form $\texttt{true}(t)$, $\texttt{does}(r, m)$, and $\texttt{distinct}(t_1, t_2)$, i. e., an atom for which no rule in D exists, or
- a formula of the form $(\exists x)f$ or $(\forall x)f$, i. e., a formula with quantified variables.

Hence, our fuzzy evaluation treats formulas with quantified variables as atomic and, thus, is essentially propositional.

We will now demonstrate with an example, how the fuzzy formula evaluation works. Recall the following rules of Tic-Tac-Toe that describe the winning condition for the `xplayer`.

```
1  line(P) :- true(cell(a,Y,P)),
2      true(cell(b,Y,P)), true(cell(c,Y,P)).
3  line(P) :- true(cell(X,1,P)),
4      true(cell(X,2,P)), true(cell(X,3,P)).
5  line(P) :- true(cell(a,1,P)),
6      true(cell(b,2,P)), true(cell(c,3,P)).
7  line(P) :- true(cell(a,3,P)),
8      true(cell(b,2,P)), true(cell(c,1,P)).
9
10 goal(xplayer,100) :- line(x).
```

The GDL formula $\texttt{goal}(\texttt{xplayer}, 100)$ describes the situation that `xplayer` wins the game. Thus, to estimate the value of a state s for the `xplayer`, we evaluate $\texttt{goal}(\texttt{xplayer}, 100)$ with respect to state s by $eval(\texttt{goal}(\texttt{xplayer}, 100), s)$.

According to Equation 5.4:

$$
\begin{aligned}
&eval(\texttt{goal}(\texttt{xplayer}, 100), s) = eval(\texttt{line}(\texttt{x}), s) \qquad (5.6)\\
&\quad = eval(\\
&\quad\ [(\exists \texttt{Y})\texttt{true}(\texttt{cell}(\texttt{a}, \texttt{Y}, \texttt{x})) \wedge \texttt{true}(\texttt{cell}(\texttt{b}, \texttt{Y}, \texttt{x})) \wedge \texttt{true}(\texttt{cell}(\texttt{c}, \texttt{Y}, \texttt{x}))] \vee\\
&\quad\ [(\exists \texttt{X})\texttt{true}(\texttt{cell}(\texttt{X}, 1, \texttt{x})) \wedge \texttt{true}(\texttt{cell}(\texttt{X}, 2, \texttt{x})) \wedge \texttt{true}(\texttt{cell}(\texttt{X}, 3, \texttt{x}))] \vee\\
&\quad\ [\texttt{true}(\texttt{cell}(\texttt{a}, 1, \texttt{x})) \wedge \texttt{true}(\texttt{cell}(\texttt{b}, 2, \texttt{x})) \wedge \texttt{true}(\texttt{cell}(\texttt{c}, 3, \texttt{x}))] \vee\\
&\quad\ [\texttt{true}(\texttt{cell}(\texttt{a}, 3, \texttt{x})) \wedge \texttt{true}(\texttt{cell}(\texttt{b}, 2, \texttt{x})) \wedge \texttt{true}(\texttt{cell}(\texttt{c}, 1, \texttt{x}))], s)
\end{aligned}
$$

Conjunctions and disjunctions in the formula are evaluated with a t-norm and t-conorm respectively (see Equation 5.1 and 5.2). Thus, Equation 5.6 above is equal to

$$
\begin{aligned}
&\perp\Big(eval([(\exists \texttt{Y})\texttt{true}(\texttt{cell}(\texttt{a}, \texttt{Y}, \texttt{x})) \wedge \ldots \wedge \texttt{true}(\texttt{cell}(\texttt{c}, \texttt{Y}, \texttt{x}))], s),\\
&\quad \perp\Big(eval([(\exists \texttt{X})\texttt{true}(\texttt{cell}(\texttt{X}, 1, \texttt{x})) \wedge \ldots \wedge \texttt{true}(\texttt{cell}(\texttt{X}, 3, \texttt{x}))], s),\\
&\qquad \ldots\\
&\qquad\ \top(\\
&\qquad\quad eval(\texttt{true}(\texttt{cell}(\texttt{b}, 2, \texttt{x})), s),\\
&\qquad\quad eval(\texttt{true}(\texttt{cell}(\texttt{c}, 1, \texttt{x})), s)\\
&\qquad\ )\\
&\qquad \ldots\\
&\quad\ \Big)\\
&\Big)
\end{aligned}
$$

The quantified formulas

$$(\exists \texttt{Y})\texttt{true}(\texttt{cell}(\texttt{a}, \texttt{Y}, \texttt{x})) \wedge \texttt{true}(\texttt{cell}(\texttt{b}, \texttt{Y}, \texttt{x})) \wedge \texttt{true}(\texttt{cell}(\texttt{c}, \texttt{Y}, \texttt{x}))$$

and

$$(\exists \texttt{X})\texttt{true}(\texttt{cell}(\texttt{X}, 1, \texttt{x})) \wedge \texttt{true}(\texttt{cell}(\texttt{X}, 2, \texttt{x})) \wedge \texttt{true}(\texttt{cell}(\texttt{X}, 3, \texttt{x}))$$

as well as the atoms

$$
\begin{aligned}
&\texttt{true}(\texttt{cell}(\texttt{a}, 1, \texttt{x})), \texttt{true}(\texttt{cell}(\texttt{b}, 2, \texttt{x})), \texttt{true}(\texttt{cell}(\texttt{c}, 3, \texttt{x})),\\
&\texttt{true}(\texttt{cell}(\texttt{a}, 3, \texttt{x})), \text{ and } \texttt{true}(\texttt{cell}(\texttt{c}, 1, \texttt{x}))
\end{aligned}
$$

are evaluated as p or $1 - p$ depending on whether or not they are entailed by the game description in combination with the state s according to Equation 5.5.

Definition 5.2 of the fuzzy formula evaluation leaves two degrees of freedom: the value to use for p and the pair of t-norm $\top$ and t-conorm $\perp$ that is used. We will discuss both in the following two sections.

5.1.2. The Value of Parameter p

The obvious value for p is $p = 1$. In this case, atoms that are true in the state s would be evaluated with 1 and atoms that are false would get a value of 0. However this choice has undesirable consequences. Consider a simple blocks world domain with three blocks a, b and c that can be stacked on each other and the goal condition

```
1 goal(player, 100) :-
2     true(on(a,b)), true(on(b,c)), true(ontable(c)).
```

The goal in this game is to build a stack of the three blocks such that a lies on top of b, b lies on top of c, and c lies directly on the table.

In this game, we want $eval(\texttt{goal}(player, 100), s)$ to somehow reflect the number of subgoals solved, i. e., the number of correctly placed blocks. However, as long as one of the atoms of the goal is not fulfilled in s, $eval(\texttt{goal}(player, 100), s)$ will be 0. The reason is that, conjunctions are evaluated with a t-norm $\top$ and $\top(0, x) = 0$ for every t-norm $\top$, which follows from the monotonicity of t-norms (see Definition 2.17 on page 16).

To overcome this problem, we use a value $p < 1$ in Definition 5.2. This solves the problem of the blocks world domain described above, as long as we use a continuous t-norm $\top$ (see Definition 2.17).

For example, consider the state $s = \{\texttt{ontable(a)}, \texttt{on(b, c)}, \texttt{ontable(c)}\}$ in which two of the three atoms of the goal condition above hold. The evaluation of the goal condition yields:

$$\begin{aligned} eval(\texttt{goal}(player, 100), s) &= eval(\texttt{true}(on(a, b)) \wedge \\ &\quad\ \texttt{true}(on(b, c)) \wedge \\ &\quad\ \texttt{true}(ontable(c)) \\ &= \top(eval(\texttt{true}(on(a, b)), s), \\ &\quad\ \top(eval(\texttt{true}(on(b, c)), s), \\ &\quad\ eval(\texttt{true}(ontable(c)), s))) \\ &= \top(1 - p, \top(p, p)) \end{aligned}$$

If $\top$ is continuous then $\top(1-p, \top(p, p)) > \top(0, \top(p, p)) = 0$. For example, using the t-norm $\top(x, y) = x * y$, we obtain $eval(\texttt{goal}(player, 100), s) = (1-p) * p^2$. A state in which only 1 of the three atoms holds would be evaluated to $(1-p)^2 * p$ which is smaller than $(1-p) * p^2$ because $p > 0.5$.

5.1.3. The Choice of T-norm and T-Conorm

Choosing a parameter $p < 1$ as discussed above introduces a new problem: Because of the monotonicity of t-norms, $eval(a_1 \wedge \ldots \wedge a_n, s) \to 0$ for $n \to \infty$. Put in words, the evaluation says that the state s is far away from a goal state in which $a_1 \wedge \ldots \wedge a_n$ holds even if all a_i already hold in s. This is a serious

problem for the comparability of different formulas. For example, consider a variation of the blocks world example:

```
goal(player, 100) :- % goal A
    true(on(a,b)), true(on(b,c)), true(ontable(c)).
goal(player, 100) :- % goal B
    true(on(block1,block2)),
    true(on(block2,block3)),
    ...
    true(on(block999,block1000)),
    true(ontable(block1000)).
```

In addition to the three blocks a, b, and c there are now another 1000 blocks. The player can get full points by either stacking a, b, and c as before ("goal A") or by stacking the other 1000 blocks on top of each other ("goal B"). Now consider the two states s_1 and s_2:

- $s_1 = \{\texttt{ontable(a)}, \texttt{on(b,c)}, \texttt{ontable(c)}, \texttt{ontable(block1)}, \ldots, \texttt{ontable(block1000)}\}$ That is, 2 of the 3 atoms of "goal A" are fulfilled but only one of the 1000 atoms of "goal B".
- $s_2 = \{\texttt{ontable(a)}, \texttt{ontable(b)}, \texttt{ontable(c)}, \texttt{on(block1,block2)}, \texttt{on(block2,block3)}, \ldots, \texttt{on(block999,block1000)}\}$ That is, only one of the 3 atoms of "goal A" are fulfilled but all of the 1000 atoms of "goal B".

For the sake of simplicity consider the t-norm $\top(x,y) = x * y$ with dual t-conorm $\bot(x,y) = x + y - x * y$. With this t-norm and t-conorm

$$\begin{aligned}
eval(\texttt{goal}(player, 100), s_1) &= \bot(p^2 * (1-p), p * (1-p)^{999}) \\
&= p^2 * (1-p) + p * (1-p)^{999} - p^3 * (1-p)^{1000} \\
&\approx p^2 * (1-p) \\
eval(\texttt{goal}(player, 100), s_2) &= \bot(p * (1-p)^2, p^{1000}) \\
&= p * (1-p)^2 + p^{1000} - p^{1001} * (1-p)^2 \\
&\approx p * (1-p)^2
\end{aligned}$$

The approximate values $p^2 * (1-p)$ and $p * (1-p)^2$ result from the fact that $p * (1-p)^{999} - p^3 * (1-p)^{1000}$ and $p^{1000} - p^{1001} * (1-p)^2$ are nearly zero. Thus, $eval(\texttt{goal}(player, 100), s_1) > eval(\texttt{goal}(player, 100), s_2)$ although the goal is already fulfilled in s_2 but not in s_1.

To mitigate this problem, we replace the t-norm $\top$ and the dual t-conorm $\bot$ in Definition 5.2 by new functions that use a threshold t with $0.5 < t \leq p$, with the following intention: values above t denote "currently true" and values below $1 - t$ denote "currently false". These new functions $\top$ and $\bot$ are defined as follows:

$$\begin{aligned}
\top(a,b) &= \begin{cases} max(\top'(a,b), t) & \text{if } min(a,b) > 0.5 \\ \top'(a,b) & \text{otherwise} \end{cases} \\
\bot(a,b) &= 1 - \top(1-a, 1-b)
\end{aligned} \tag{5.7}$$

where $\top'$ denotes an arbitrary t-norm.

These functions ensure that formulas that are true wrt. state s are always evaluated to a value greater or equal t and formulas that are false are assigned a value smaller or equal $1-t$ by the function *eval*. Thus the values of different formulas stay comparable. The disadvantage is that $\top$ is not associative, i. e., $\top(x_1, \top(x_2, x_3)) = \top(\top(x_1, x_2), x_3)$ may not hold, at least in cases of continuous t-norms $\top'$. Thus, $\top$ is not a t-norm itself in general.

Loosing associativity means that the evaluation of formulas such as $eval(f_1 \wedge (f_2 \wedge f_3), s)$ and $eval((f_1 \wedge f_2) \wedge f_3, s)$ can yield different values. However, the problem that semantically equivalent but syntactically different formulas are evaluated differently already exists without our new definition of $\top$ and $\bot$. For example, $eval(a \wedge a, s)$ and $eval(a, s)$ are only equal in case $\top$ is idempotent. This is only the case for the t-norm $\top(x, y) = min(x, y)$ [KGK95] which is not continuous. Furthermore, by choosing an appropriate t-norm $\top'$ that is used in Equation 5.7, it is possible to minimize this effect.

For the t-norm $\top'$ we use an instance of the Yager family of t-norms [KGK95]:

$$\begin{aligned} \top'(a,b) &= 1 - \bot'(1-a, 1-b) \\ \bot'(a,b) &= (a^q + b^q)^{\frac{1}{q}} \end{aligned}$$

The Yager family captures a wide range of different t-norms. Ideally, we want a heuristic that is able to differentiate between all states that are different with respect to the goal and terminal formulas. By varying q, in the t-norm described above, one can choose an appropriate t-norm for each game, depending on the structure of the goal and terminal formulas to be evaluated, such that as many states as possible that are different with respect to the goal and terminal formulas are assigned a different value by the heuristic function.

Larger values for q reduce the problem of $\top'(a, b)$ falling below the threshold t in equation 5.7. However, if q is chosen too large the t-norm degenerates to a non-continuous function because $\bot'(a, b) = (a^q + b^q)^{\frac{1}{q}} \approx max(a, b)$ for large q. In that case the evaluation of a formula degenerates to a binary evaluation where $eval(f, s) = p$ if f holds in s and $1 - p$ otherwise. We want to avoid this because then the fuzzy evaluation cannot distinguish between formulas that are fulfilled to a different degree.

In our current implementation, we use the hand-crafted values $p = 0.75$, $t = 0.55$, and $q = 15$ for all games. In principle, it is possible to choose values based on the size of the formulas that have to be evaluated. However, our fixed values proved to work well with all game descriptions that are currently available, that is, all states that are different with respect to the goal or terminal condition get different heuristic values. With these values for p, q and t, the case that $eval(f_1 \wedge \ldots \wedge f_n, s) = t$, where $f_1, \ldots, f_n$ hold in s, only occurs for $n > 6746$ $(1 - (n * (1-p)^q)^{\frac{1}{q}} >= t)$. Thus, for conjunctions with at most 6746 conjuncts the case that the value of a conjunction, i. e., the Yager t-norm, falls below the threshold does not occur. Formulas of this size do not occur in any of the games that were played at GGP competitions so far.

5.1.4. Theoretical Properties of the Fuzzy Evaluation

According to Definition 5.2 and the definitions of $\top$ and $\bot$ above, the fuzzy formula evaluation $eval(f, s)$ has the property that its value is greater than 0.5 if and only if the formula f holds in state s. This property shows the connection between the fuzzy evaluation and the logical formula it evaluates.

Theorem 5.1. *Let D be a valid game description, s be a state of a game corresponding to D and f be a GDL formula.*

$$eval(f, s) \geq t > 0.5 \quad iff \quad D \cup s^{\texttt{true}} \models f$$
$$eval(f, s) \leq 1 - t < 0.5 \quad iff \quad D \cup s^{\texttt{true}} \models \neg f$$

Proof. This property can be easily proved by induction over the size of the GDL formula f. For formulas f for which Equation 5.5 applies, the property follows immediately from the equation itself together with the fact that $p > 0.5$.

Now, assume the property holds for formulas f and g.

- From Equations 5.7 and 5.1, it follows that the property holds for $f \wedge g$.
- Similarily, it follows from Equations 5.7 and 5.2 that the property holds for $f \vee g$.
- If f holds in s, $eval(f, s) \geq t$. Thus, $\neg f$ does not holds in s and $eval(\neg f, s) = 1 - eval(f, s) \leq 1 - t$. The case where f does not hold in s is proved in the same way. Hence, the property holds for $\neg f$.

□

As we can see, the evaluation of a formula corresponds to the truth value of the formula in the sense that, $eval(f, s) > 0.5$ if and only if the formula f holds in the state s.

5.1.5. State Evaluation Function

Up to now, we only defined the evaluation of formulas with respect to a state. We said that in order to evaluate whether a state s is advantageous for a player r we would use the fuzzy evaluation $eval(f, s)$ of formula f, where f describes the situation that r wins the game. However, the game description language allows to describe games with different "levels" of winning: the goal value of a player can be any value in the range $[0, 100]$. Thus, there can be different goals in the game for a player that result in different rewards. For example, Tic-Tac-Toe describes three different goals:

- a player obtains 100 points for building a line of three of its symbols,
- each player gets 50 points if none of the players was able to build a line of three, and
- a player gets 0 points if its opponent gets 100 points.

Other games define even more different goal values. For example in Skirmish, each player gets a score that is proportional to the number of pieces it captured.

We take all these different goals into account by averaging over the evaluations of the formulas for each of the goals. We define the state evaluation function $h(r, s)$ (heuristics) for a role r in state s in a particular game based on the evaluation of the terminal condition and the goal conditions for role r as follows:

Definition 5.3 (State Evaluation Function $h(r, s)$)**.** *Let GV be the domain of goal values, i. e., the set of all possible goal values for the game.*

$$
\begin{aligned}
h(r,s) &= \frac{100}{\sum_{gv \in GV} gv} * \sum_{gv \in GV} gv * h(r, gv, s) \\
h(r,gv,s) &= \begin{cases} eval(\texttt{goal}(r,gv) \vee \texttt{terminal}, s) & \text{if } D \cup s^{\texttt{true}} \models \texttt{goal}(r,gv) \\ eval(\texttt{goal}(r,gv) \wedge \neg\texttt{terminal}, s) & \text{else} \end{cases}
\end{aligned}
$$

Thus, we compute the heuristics value $h(r, s)$ of a state s for role r by a linear combination of the heuristics $h(r, gv, s)$ for each goal value gv of the domain of goal values GV weighted by the goal value gv.

If the goal is already fulfilled (but the terminal state is not reached yet), we compute the heuristics $h(r, gv, s)$ as $eval(\texttt{goal}(r, gv) \vee \texttt{terminal}, s)$, that is, as the evaluation of the disjunction of the goal and terminal formulas. Thus, the heuristics ensures that the player tries to reach a terminal state if the goal is reached because

$$eval(\texttt{goal}(r,gv) \vee \texttt{terminal}, s) = \bot(eval(\texttt{goal}(r,gv), s), eval(\texttt{terminal}, s))$$

and $\bot$ is monotonically increasing in both arguments. Hence, between two states that both fulfil $\texttt{goal}(r, gv)$ to the same degree, we prefer the one that fulfils `terminal` to the higher degree, if the goal is already reached.

If the goal is not reached yet (the "else" case of $h(r, gv, s)$), we compute $h(r, gv, s)$ as $eval(\texttt{goal}(r, gv) \wedge \neg\texttt{terminal}, s)$, that is, as the conjunction of the goal and the negated terminal formulas. This heuristics tries to avoid terminal states as long as the goal is not reached because

$$eval(\texttt{goal}(r,gv) \wedge \neg\texttt{terminal}, s) = \top(eval(\texttt{goal}(r,gv), s), 1 - eval(\texttt{terminal}, s))$$

and $\top$ is monotonically increasing in both arguments. Hence, between two states that both fulfil $\texttt{goal}(r, gv)$ to the same degree, we prefer the one that fulfils `terminal` to the lower degree, if the goal is not reached yet.

The disjunction $\vee$ is used in the first case and conjunction $\wedge$ in the second case in order to ensure that a state s_1 in which the goal is already fulfilled, gets a higher heuristic value than a state s_2 in which the goal does not hold:

$$
\begin{aligned}
eval(\texttt{goal}(r,gv) \vee \texttt{terminal}, s_1) &> 0.5 \\
eval(\texttt{goal}(r,gv) \wedge \neg\texttt{terminal}, s_2) &< 0.5
\end{aligned}
$$

We use the heuristics $h(r, s)$ during search to evaluate non-terminal leaf nodes in the search tree (see Definition 4.2 on page 32). Our approach allows us to base the evaluation of non-terminal states directly on the goal definition without the need of using learning techniques. In Fluxplayer, we use the start clock of a match, i. e., the phase between start message and first play message (see Section 3.2), to generate Prolog code that computes $h(r, s)$ from the goal and terminal rules of the game description. This task takes time in the order of milliseconds, even for complex games.

5.2. Identifying Structures in the Game

In this section we describe how we can identify additional structures in the game, such as a game board, that can be used to improve the state evaluation function presented in the previous section.

The fuzzy state evaluation described above evaluates atoms of the form `true`(f), i. e., atoms the refer to a fluent in the current state, with the fixed values p or $1 - p$, depending on whether the fluent holds in the current state or not. This seems to be a feasible approach if the fluent f encodes an essentially binary information, such as, `control(xplayer)` in Tic-Tac-Toe, which either holds (if it is `xplayer`'s turn) or not (if it is `oplayer`'s turn). However, some fluents encode information that gives rise to a more fine grained evaluation.

For example, consider a simple racing game with one piece that can be moved 1, 2 or 3 squares at a time and is initially located at square 1. The current square of the piece shall be described by the fluent `square(P)`, e. g., `square(1)` in the initial state. Now consider the goal condition `true(square(10))`, that is, the goal is to reach square 10. Our heuristic evaluation function defined in the previous section cannot distinguish between, e. g., the initial state and a state where the piece is at square 9. In both cases the atom `true(square(10))` will be evaluated to $1 - p$ by Equation 5.5. However, intuitively the state where the piece is at square 9 should be preferred because it is only one step away from the goal. Instead of evaluating `true(square(10))` with the value p or $1 - p$ in Equation 5.5 depending on whether `true(square(10))` holds or not, the evaluation function should reflect the distance of the current location of the piece to square 10. However, in order to compute such a distance, it is necessary to have more information about the game, for example, the order in which the squares are laid out.

Of course, in General Game Playing, we have to obtain distance information in an automated fashion. In the following sections, we present an approach to analyse games in order to find structures, such as the game board in the example, that can be used to compute distances in order to improve the evaluation function. Our approach is similar to the one described by Kuhlmann et. al. in [KDS06] but more general in two ways:

- Kuhlmann et. al. heavily rely on syntactic structure of the game rules to identify certain structures in the game because they use pattern matching

on the game rules. Instead, we use the semantics of predicates in the game rules. Thus, we can identify structures independently of a fixed syntax. We can also identify structures whose game rules can have too many syntactical variations for effectively using pattern matching to identify the structures.
- Wherever possible, we find structures independently of the arity of the fluents of the game. For example, we can identify n-dimensional boards for arbitrary n, where the method in [KDS06] is limited to two dimensions.

In the following sections we first present algorithms for identifying certain structures in the game:

- In Section 5.2.1, we present an algorithm for computing the **domains of relations and functions** in the game description, i. e., the sets of possible values that arguments of the relations and functions can take. This is a prerequisite for the remaining sections.
- Section 5.2.2 describes how we identify **static structures** in the game, i. e., structures that are independent of the game state. This includes structures, such as, successor relations or orders.
- Section 5.2.3 describes how we identify **dynamic structures**, i. e., structures in the fluents of the game state. The most prominent example for dynamic structures is a game board.

We will describe how we use the identified structures to improve the evaluation function in Section 5.3.

5.2.1. Domains of Relations and Functions

For the identification of structures, it is necessary to know the domains of relations and functions, i. e., the range of possible values of the arguments of relations and functions of the game description. Additionally, it is helpful to know which relations and functions are somehow connected by the game rules, that is, which of them have the same domain.

For example, to know that the distance between `square(1)` and `square(10)` in our simple racing game is 9, we need to find out that the values in the argument of the fluent `square` are ordered according to the order: $1 \mapsto 2 \mapsto \ldots \mapsto 10$. Often, orders are defined by additional predicates in the game description. In our racing game, this could be a successor relation `succ` with the following rules:

```
1 succ(1,2).
2 succ(2,3).
3 ...
4 succ(9,10).
```

To find an order for the domain of the argument of `square`, we can look for a predicate in the game that defines an order and has the same domain as the fluent `square`. The predicate `succ` above would be such a predicate.

Furthermore, we need to normalise the distance, i. e., map it to values in the range $[0, 1]$, because our evaluation function is only defined for values in this

range (see Definition 5.2). Therefore, we need to know the maximal possible distance. In our example, this is the number of squares in the game, which is the same as the size of the domain of the argument of the fluent `square`.

We can obtain both information, the domains of functions and relations and which of the functions and relations are connected by the game rules, with the help of the algorithm that we present in the following.

We compute the domains, or rather supersets of the domains, of all relations and functions of the game description by generating a *domain graph* from the rules of the game description. Figure 5.1 shows the domain graph for the following game rules describing an ordinary step counter:

```
1 succ(0, 1).
2 succ(1, 2).
3 succ(2, 3).
4 init(step(0)).
5 next(step(X)) :-
6     true(step(Y)),
7     succ(Y, X)).
```

A domain graph contains the following nodes:

- There is a node `p,i` in the graph for every argument position $i \in \{1, \ldots, n\}$ of each n-ary symbol p in the game description. This applies to both function and predicate symbols.

  In Figure 5.1, these argument position nodes are depicted by ellipses and include, among others, the nodes `succ,1` and `succ,2` referring to the two arguments of the binary predicate *succ* and the node `step,1` referring to the first and only argument of the unary function *step*.

- There is a node `f/n` for each n-ary function symbol f, where constants are treated as nullary function symbols.

  In Figure 5.1, the nodes of this type are `0/0`, `1/0`, `2/0`, and `3/0` for the constants of the respective names and the node `step/1` for the unary function symbol *step*. These nodes are depicted by rectangles.

The edges of a domain graph are defined as follows:

- There is an edge between an argument node `p,i` and a function symbol node if there is a rule in the game where the function symbol appears as the i-th argument of a function or predicate with name p.

  For example, there is an edge between the nodes `step,1` and `0/0` because 0 occurs in the first argument of *step* in the rule `init(step(0))`.

- There is an edge between two argument position nodes `p,i` and `q,j` if there is a rule in the game in which the same variable appears as the i-th argument of p and the j-th argument of q.

  In our example, there are two variables `X` and `Y`. The variable `X` occurs as the first argument of *step* and as the second argument of *succ* in the

last rule. Hence, the nodes `step,1` and `succ,2` are connected. Similarly, `step,1` and `succ,1` are connected because of `Y` in the same rule.

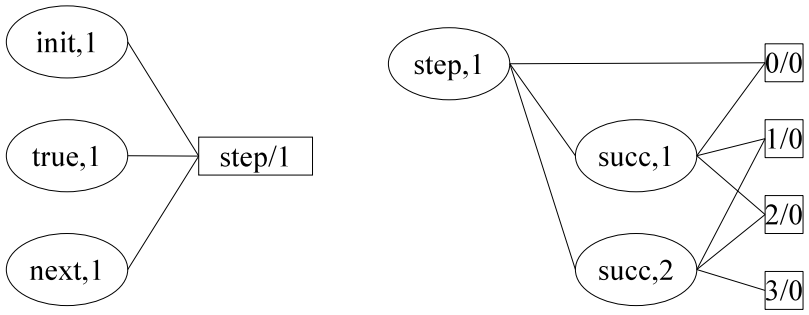

Figure 5.1.: An example domain graph for calculating domains of functions and predicates. Ellipses denote arguments of functions or predicates and squares denote constants or function symbols.

The argument positions in each connected component of the domain graph share a domain. The constants and function symbols in the connected component are the domain elements. Thus, to compute domains of functions and relations of a game description, we compute the connected components of the domain graph described above.

We denote the domain of the i-th argument of a function or predicate p with $dom(p, i)$.

In our example, we obtain the following domains:

$$\begin{aligned} dom(step, 1) = dom(succ, 1) = dom(succ, 2) &= \{0, 1, 2, 3\} \\ dom(init, 1) = dom(next, 1) = dom(true, 1) &= \{step/1\} \end{aligned}$$

The computed domain $\{0, 1, 2, 3\}$ is actually a superset of the real domain of the arguments of `succ`. For example, `3` cannot occur in the first argument of `succ`. We disregard this fact because we are more interested in the dependencies between different functions and predicates than in the actual set of possible values.

We will use the information about the domains to improve our state evaluation function in the following sections. Furthermore, we use domain information to (partially) ground game descriptions in order to improve certain game analyses (see Chapter 6). We also need domain information for proving properties of games (see Chapter 7).

5.2.2. Static Structures

By static structures of games we mean predicates of the game description that have special properties. We call these predicates static structures because they do not depend on the state of the game. We detect the following types of static structures:

successor relation A successor relation is a binary relation that defines a successor function over a finite set of terms. An example is the binary relation `succ` in the example above, which defines 1 to be the successor of 0, 2 to be the successor of 1, etc. We consider all *binary relations* that are *antisymmetric, functional* and *injective* as successor relations.

order relation An order relation is a binary relation that defines a total or partial order of a finite set of terms. Examples are the usual arithmetic relations $<$ and $>$ over numbers. Order relations are often defined as the transitive closure of a successor relation. We consider all *binary relations* that are *antisymmetric* and *transitive* as order relations. Thus, we consider both strict and non-strict partial orders.

Unlike Kuhlmann et. al. in [KDS06], who use the syntactical structure of the rules, we exploit semantical properties of the predicates to detect static structures. Because we use semantical instead of syntactical properties, we can also detect higher-level predicates like order relations. This is difficult to do when relying on the syntax of the rules because there are many semantically equivalent but syntactically different descriptions of a predicate. The properties of the predicates can be proved easily because all domains are finite as a result of the restrictions on a game description in GDL (Definition 2.7).

We detect successor and order relations in the following way. For every binary relation r in the game description D we follow these steps:

1. Compute the domains $dom(r, 1)$ and $dom(r, 2)$ of both arguments of r using the algorithm described in Section 5.2.1.
2. If $dom(r, 1) \neq dom(r, 2)$, we do not consider r as successor or order relation, otherwise
3. If for all $x, y, z \in dom(r, 1)$, the two properties

$$\begin{aligned} D \models\ & r(x,y) \wedge r(y,x) \supset x = y && \text{(antisymmetry) and} \\ D \models\ & r(x,y) \wedge r(y,z) \supset r(x,z) && \text{(transitivity)} \end{aligned}$$

   hold, then r is an order relation.
4. If for all $x, y, z \in dom(r, 1)$, the three properties

$$\begin{aligned} D \models\ & r(x,y) \wedge r(y,x) \supset x = y && \text{(antisymmetry),} \\ D \models\ & r(x,y) \wedge r(x,z) \supset y = z && \text{(functional), and} \\ D \models\ & r(y,x) \wedge r(z,x) \supset y = z && \text{(injective)} \end{aligned}$$

   hold, then r is a successor relation.

5.2.3. Dynamic Structures

Dynamic structures are structures that change with the progress of the game. Thus, they are somehow connected to the game state. We consider two types of dynamic structures, game boards and quantities.

Game Boards For us a game board is a multi-dimensional grid of cells that have a state that can change. We consider each fluent in the game description with at least 2 arguments as a potential board. Some of the arguments must identify the cell of the board, those are the coordinates. The remaining arguments form the cell's state or content. They must have a unique value for each instance of the coordinate arguments in each state of the game. That is, every cell of the board has just one content. A few game descriptions with multiple contents of a board's cell exist. For example, in Pacman the ghosts may be at the same location on the board as Pacman. However, we choose to ignore these cases in order to have an easy way to differentiate between coordinate arguments and those referring to contents of a cell. We call the coordinates of the cell the *input arguments* and the remaining arguments the *output arguments* of the fluent that describes the board.

A board is ordered if the domains of some of the input arguments are ordered, i. e., there are successor or order relations for some of the domains of the input arguments. If not all of the coordinate arguments are ordered, then the fluent possibly describes multiple ordered boards. An example for this case is the fluent `cell(B, Y, X, C)` in the Racetrack game (the final game in the AAAI GGP Competition 2005), a two-player racing game where each player moves on his own board. The arguments `Y` and `X` are the coordinates of the cell, while `B` identifies the board. The argument `C` is the content of the cell. Only the domains of `X` and `Y` are ordered, but the input arguments are `B`, `X`, and `Y`.

We use the information about game boards to compute distances between the current and the goal location of pieces on the board. These distances are then used to evaluate atoms of the goal formula. The evaluation is described in detail in Section 5.3.3.

To actually detect game boards we need to distinguish between input and output arguments of a fluent. We will show how to do this after explaining the other type of dynamic structure that we are interested in, that is, quantities.

Quantities In our setting, the term quantity refers to fluents describing a quantity or amount of something. For example, the fluent `pieces(Player,Number)` of the game Checkers, which describes the number of pieces each player has left in a certain state describes a quantity, or rather a set of quantities, one for each player. For us a fluent is a quantity if it has an ordered output argument. Hence, a quantity is a special case of a game board where the coordinates are not necessarily ordered but the contents of the cells are. For example, the fluent `pieces(Player,Number)` describes one-dimensional unordered board with one cell for each player where the content of each cell is the amount of pieces the player has left. The first argument `Player` is the coordinate of the cell and the second argument `Number` the content. Because the output argument (`Number`) is ordered, `pieces(Player,Number)` describes a quantity.

Also unary fluents can describe quantities. A prominent example is the step counter `step(X)` occurring in many games. The step counter describes a quantity, namely the number of steps executed so far in the game. A unary fluent $f(x)$

describes a quantity if it refers to a null-dimensional board with an ordered output argument x. Hence, the fluent is a singleton, i. e., for every game state s there is at most one instance of x such that $f(x) \in s$.

Another example for a quantity is the fluent `square(X)` denoting the location of the only piece in the simple racing game described at the beginning of Section 5.2. Note, that we detect `square(X)` as a quantity, although the intention of the game designer was to encode the location of a piece on a board. This shows, that the terms quantity and game board that we use here are not necessarily connected to actual quantities or boards in the game but rather stand for certain properties that a fluent of a game displays.

Input and Output Arguments We defined game boards and quantities by referring to the input and output arguments of fluents. Hence, for deciding if a fluent describes a board or a quantity, we need to find the fluent's input and output arguments.

Definition 5.4 (Input and Output Arguments)**.** *Consider a game with ground terms Σ and reachable states $\mathcal{S}_r$ (see Definition 2.4). Let f be an n-ary function symbol in the game, and $I \subseteq \{1, 2, \ldots, n\}$ be a subset of the argument indices of f, and $O = \{1, 2, \ldots, n\} \setminus I$ be the set of remaining argument indices. The set I is called a set of input arguments of the fluent f if and only if I is the smallest set with respect to set inclusion such that*

$$f(t_1, \ldots, t_n) \in s \wedge f(t'_1, \ldots, t'_n) \in s \wedge ((\forall i \in I) t_i = t'_i) \supset (\forall o \in O) t_o = t'_o \tag{5.8}$$

for all reachable states $s \in \mathcal{S}_r$, and ground terms $t_1, \ldots, t_n, t'_1, \ldots, t'_n \in \Sigma$.

The set O is called a set of output arguments of f.

The intuitive meaning of this definition is that for each instance of the input arguments of a fluent there is at most one instance for the output arguments such that the fluent holds in a state.

This definition implies that the set of all argument indices $\{1, \ldots, n\}$ is a set of input arguments for any n-ary fluent. However, this set is not interesting. After all, we want separate the coordinates and contents of boards by detecting input and output arguments, but if all arguments are input arguments, no output arguments, i. e., content of the board's cells, are left. Thus, we want to compute the minimal sets of input arguments with respect to set inclusion.

Definition 5.5 (Minimal Set of Input Arguments)**.** *Let I be a set of input arguments for a fluent f of a game. I is a minimal set of input arguments if and only if there is no set of input arguments I' of f such that $I' \subset I$.*

Note that there may be different minimal sets of input arguments for the same fluent. For example, consider a game where each player picks a different colour. The fluent `picked(Player, Color)` shall describe which player picked which colour. Because a player can have only one colour, $I = \{1\}$ is a set of input

arguments for `picked`, i. e., `Player` is an input argument. However, every colour can only be picked by one player. Consequently, $I' = \{2\}$ (`Color`) is also a set of input arguments and neither set is a subset of the other.

Input and output arguments can be computed in different ways. We have implemented two methods. The first one is similar to the method described in [KDS06]. There, input and output arguments of all fluents are approximated by generating hypotheses and checking them in states generated by random play until the hypotheses are stable, i. e., until the hypotheses were verified by a large enough number of states without finding counter examples. Usually, a few hundred states are sufficient to be relatively certain that the hypotheses are correct. However, this algorithm does not give any guarantees unless the hypotheses are checked in all reachable states.

Our first algorithm for detecting sets of input arguments for a fluent intuitively works as follows. Since we want to compute minimal sets of input arguments, we start by with the hypothesis that the empty set is a set of input arguments. Then we check this hypothesis with randomly generated reachable states. If the hypothesis is refuted we add an additional argument index to it. We keep doing this until we find a hypothesis that is not refuted by m newly generated states, where m is some predefined threshold. We define the following abbreviations to improve the readability of our algorithm:

$$\mathcal{I}^{-I} \stackrel{\text{def}}{=} \mathcal{I} \setminus \{I\} \text{ (the set of hypotheses } \mathcal{I} \text{ without the hypothesis } I)$$
$$I^{+i} \stackrel{\text{def}}{=} I \cup \{i\} \text{ (add argument index } i \text{ to the hypothesis } I)$$

Then we define the algorithm to compute the minimal sets of input arguments for an n-ary fluent f as follows:

1. The initial set of hypotheses for sets of input arguments of f is $\mathcal{I} = \{\emptyset\}$. Thus, the only first hypothesis is that the set of input arguments is empty. This hypothesis holds if there is at most one fluent with name f in each state.
2. Generate a random state s that is reachable from the initial state of the game.
3. For every $I \in \mathcal{I}$ check whether the equation 5.8 holds. If not, set

$$\mathcal{I} := \mathcal{I}^{-I} \cup \{I^{+i} \mid i \in 1 \ldots n \ \wedge \ i \notin I \ \wedge \ \neg(\exists I' \in \mathcal{I}^{-I}) I' \subset I^{+i}\}$$

   Hence, we remove the hypothesis I that turned out wrong and add new hypotheses I^{+i} that are derived from the removed hypothesis I by adding one additional argument index i. However, we only add a new hypotheses if there is no smaller hypothesis (wrt. set inclusion) left that was not refuted yet.
   For example, consider the ternary fluent $f(x, y, z)$ and the set of hypotheses $\mathcal{I} = \{\{1\}, \{3\}\}$, that is, the hypotheses $\emptyset$ and $\{2\}$ were already refuted. Now, let the first hypothesis $I = \{1\}$ be refuted, that is, there is a state s such that there is more than one instance of $f(x, y, z)$ in s for the same x. In that case we remove $I = \{1\}$ and generate the new hypotheses $\{1, 2\}$

and $\{1, 3\}$, that is, the hypothesis that are derived from I by adding and additional argument index of f. However, only $\{1, 2\}$ is added as a new hypothesis because $\{3\}$ is a subset of $\{1, 3\}$ and not refuted yet. Thus, the set of hypotheses for the next iteration is $\mathcal{I} = \{\{1, 2\}, \{3\}\}$. Note, that the hypothesis $\{1, 3\}$ will be added once hypothesis $\{3\}$ is refuted.

4. If all hypotheses in $\mathcal{I}$ were affirmed for at least m states or $\{1, \ldots, n\} \in \mathcal{I}$, return $\mathcal{I}$ as sets of input arguments. Otherwise, go to 2.

This algorithm computes the minimal input arguments for one fluent and has to be run for each fluent for which we want to know the input arguments. Of course, the hypotheses for all fluents of a game can be checked in parallel. Thus we can reuse the states generated in step 2. At the same time, this simulation stage can also be used to check other properties of the game, e. g., if the game is turn-based.

Usually, the hypotheses are stable and correct after a small number of iterations (in the order of tens or hundreds). The disadvantage of this method is, that the result is not necessarily correct unless all reachable states of the game are checked. If the found properties (i. e., input and output arguments of fluents) are only used in the state evaluation function, it is acceptable to get incorrect results sometimes. In this case, the player might play sub-optimal moves, if the evaluation function is affected by accepting wrong hypotheses. However, another potential application of the discovered properties is to transform the game description into a form that is more efficient for reasoning (cf. [KE09]). In that case, accepting wrong hypothesis may cause the player to play illegal moves or crash.

In Chapter 7, we present a proof system for general games that is able to proof the correctness of an hypothesis for a set of input arguments. This is a second method to compute input and output arguments of fluents. It has the advantage of providing correct results, but may fail for complex games.

In Fluxplayer, we use a combination of both approaches: We run the above algorithm with a small threshold m (5) to compute hypotheses and then try to prove the correctness of these hypotheses with the method described in Chapter 7.

5.3. Using Identified Structures for the Heuristics

The state evaluation function introduced in Section 5.1 uses a binary evaluation of atomic sub-formulas of the goal and terminal condition (Definition 5.2, Equation 5.5):

$$eval(f, s) = \begin{cases} p & \text{if } D \cup s^{\texttt{true}} \models (\exists) f \\ 1 - p & \text{otherwise} \end{cases}$$

Thus, a formula f is evaluated with p if it holds in state s and with $1 - p$ otherwise.

If the formula f refers to an identified structure of the game, e. g., a game board or a quantity, we will replace this binary evaluation with a distance function that yields a more fine grained evaluation and, thus, can distinguish between more states. This distance function computes an estimate of the distance from the state s to a state in which f holds by taking the identified structure into account. For example, if $f =$`true(square(10))` in the simple racing game that we introduced in Section 5.2, then the distance function will reflect the distance between the current location of the piece and square 10. Thus, $eval(f, s)$ will yield a higher value for a state s_1 in which `true(square(9))` holds, than in a state s_2 in which `true(square(3))` holds, i. e., where the piece has only been moved to square 3. The current evaluation function cannot distinguish between states s_1 and s_2 because in both cases f does not hold.

In the following, we replace Equation 5.5 in Definition 5.2 with functions presented in the following sections, if f refers to one of the identified structures, namely, order relations, game boards, or quantities.

5.3.1. Order Relations

We define the fuzzy evaluation of an order relation $r(a, b)$ as follows:

$$eval(r(a,b),s) = \begin{cases} t + (1-t) * \frac{\Delta_r(a,b)}{|dom(r,1)|} & \text{if } D \cup s^{\texttt{true}} \models r(a,b) \\ (1-t) * (1 - \frac{\Delta_r(b,a)}{|dom(r,1)|}) & \text{if } D \cup s^{\texttt{true}} \models r(b,a) \\ 0, \text{otherwise} \end{cases}$$

Here, $\Delta_r(a, b)$ is the distance between a and b according to r and $|dom(r, 1)|$ denotes the size of the domain of the arguments of r (remember that both arguments have the same domain, i. e., $dom(r, 1) = dom(r, 2)$). Thus, the evaluation function computes the normalized distance between a and b in the following way:

- If $r(a, b)$ holds in the state s, the distance is mapped into the interval $[t, 1[$ with higher distances yielding higher values.
- If $r(b, a)$ holds in the state s, $r(a, b)$ can not be fulfilled. Thus, the distance is mapped into the interval $]0, 1 - t]$ with higher distances yielding lower values.
- Because r may be a partial order, a and b could be incomparable, i. e., neither $r(a, b)$ nor $r(b, a)$ holds. In this case, we assume that it is not possible to fulfil $r(a, b)$ and assign the value 0.

We compute $\Delta_r(a, b)$ as the shortest path from a to b in the Hasse diagram of r. The Hasse diagram is a directed graph $G = (V, E)$, where $V = dom(r, 1) = dom(r, 2)$ and $E = \{(a, b) | r(a, b) \wedge \neg(\exists c) r(a, c) \wedge r(c, b)\}$. Hence, the vertices of the graph G are the ground terms in the domain of r. Furthermore, G contains an edge from a to b whenever a is "smaller" than b according to r and there is no element between a and b. Thus, the Hasse diagram reflects a successor relation between elements of the domain and can be used to count the number

of steps between two elements of the domain by computing the length of the path between the two elements in the Hasse diagram.

This new fuzzy evaluation function has the following advantage over the previous binary evaluation: It prefers states with a narrow miss of the goal over states with a strong miss. The same holds for states on the winning side, where the evaluation function prefers states with a strong winning margin over states with a small one. Let us explain this with the following example.

Othello is a two-player game with the objective to have more pieces of the own colour on the board than the opponent. The goal condition for the red player of Othello is:

```
1 goal(red, 100) :-
2     gameover,
3     tally(B, R),
4     lessthan(B, R).
```

The predicate `gameover` is fulfilled if a terminal state of the game is reached and `tally(B, R)` holds if there are `B` black pieces and `R` red pieces on the board. The predicate `lessthan(B, R)` is an order relation according to our definition. In this game, a state with `R=20` and `B=10` will get a higher value for the red player than a state with `R=16` and `B=14`. Thus, the new evaluation function prefers states with a stronger winning margin. This is desirable because the chance of winning the game is typically higher for red if the gap between `R` and `B` is bigger in an intermediate state.

5.3.2. Quantities

To take advantage of the identified quantities in a game, we replace the fuzzy formula evaluation (Equation 5.5 in Definition 5.2) for atoms of the form `true(f)` if the `f` in question describes a quantity. If the fluent `f` is a quantity, we use the difference between the quantity in the current state and the quantity in `f` as basis of the evaluation of `true(f)`.

We define the evaluation of quantities, as well as game boards, in terms of a distance function $\delta(s, f)$ between a state s and a fluent f. The intended meaning is that the distance is proportional to the number of steps needed to reach a arbitrary state s' from s such that f holds in s'. Thus, the minimal distance is $\delta(s, f) = 0$ if $f \in s$, i. e., f already holds in s. Furthermore, the distance shall be infinite if no state s' with $f \in s'$ is reachable from s. We define the distance to be normalized in the interval $]0, 1[$ in all other cases. This ensures that we can easily use it in our evaluation function directly.

With the help of this distance function we can easily define the evaluation of quantities as follows:

$$eval(\texttt{true}(f), s) = \begin{cases} p & \text{if } D \cup s^{\texttt{true}} \models \texttt{true}(f) \\ 0 & \text{if } \delta(s, f) = \infty \\ (1 - t) * (1 - \delta(s, f)) & \text{otherwise} \end{cases} \tag{5.9}$$

Thus, we consider the following three cases:

- An atom $\texttt{true}(f)$ is evaluated to the constant p, if it holds in the state s, as before.
- If the distance $\delta(s, f)$ is infinite, then $eval(\texttt{true}(f), s) = 0$ to indicate that there is no way of fulfilling $\texttt{true}(f)$ once the game has reached state s.
- If the distance is finite, it is mapped into the interval $]0, 1 - t]$, i. e., the interval for "false" values. We do this mapping in such a way that a distance near 0 yields an evaluation near $1 - t$ and the maximal distance yields a value near 0, that is, higher distances cause lower values in the evaluation. The constant t is the threshold t, that we introduced in Section 5.1.3.

The question remains, how we define the distance function $\delta(s, f)$ based on the information that we found about the quantity f. For the sake of simplicity, let $f(x, y)$ be a binary fluent describing a quantity y for each instance of x. Hence, f has the set of input arguments $\{1\}$, and the domain of the second argument of f – the output argument – is ordered. Let r be the order relation for the domain of the second argument of f. We compute the distance from a state s to a ground instance of $f(x, y)$ as follows:

$$\delta(s, f(x, y)) = \begin{cases} \frac{\Delta_r(y,y')}{|dom(f,2)|} & \text{such that } f(x, y') \in s \\ \infty & \text{if } \neg(\exists y') f(x, y') \in s \end{cases} \tag{5.10}$$

Thus, we estimate the number of steps needed to fulfil $f(x, y)$ by finding a fluent $f(x, y')$ in s and computing the difference $\Delta_r(y, y')$ between y and y' according to the order relation r of the domain of y. The distance is then computed by normalising this difference $\Delta_r(y, y')$ to the interval $[0, 1[$ by dividing with the size of the domain of y ($|dom(f, 2)|$). Note that there is at most one instance of y' for any given x such that $f(x, y') \in s$ because x is the input argument of f. However, the definition of input arguments (Definition 5.4) does not require that there exists a fluent $f(x, y')$ for every x. In this case, we define the distance to be maximal (∞).

For an example, we can apply the improved evaluation function together with the distance function (Equation 5.10) to the game of Checkers, directly. The goal condition for the white player in Checkers is that there are no black pieces left: `true(pieces(black,0))`. The fluent `pieces` is identified as a quantity with input arguments $\{1\}$ and ordered output arguments $\{2\}$. Thus, the state evaluation function for the white player reflects the difference of the current number of black pieces to zero and is lower, the more black pieces are still left in the current state.

Although the distance function in Equation 5.10 only applies to binary fluents, we can easily extent it to an arbitrary number of input and output arguments in the following way. Let $f(\vec{x}, \vec{y})$ be a fluent describing a quantity, where $\vec{x} = (x_1, \ldots, x_n)$ denotes the input arguments of f and $\vec{y} = (y_1, \ldots, y_m)$ denotes the output arguments. Furthermore, let $r_1, \ldots, r_m$ be order relations for the respective output arguments $y_1, \ldots, y_m$ of f. We compute the distance from

state s to $f(\vec{x},\vec{y})$ as follows:

$$\delta(s, f(\vec{x},\vec{y})) = \begin{cases} \frac{1}{m} * \sum_i \frac{\Delta_{r_i}(y_i,y_i')}{|dom(f,n+i)|} & \text{such that } f(\vec{x},\vec{y'}) \in s \\ 1 & \text{if } \neg(\exists \vec{y'}) f(\vec{x},\vec{y'}) \in s \end{cases}$$

With $|dom(f, n+i)|$ we denote the size of the domain of the $(n+i)$-th argument of f, i. e., the i-th output argument of f.

5.3.3. Game Boards

The evaluation of game boards can be defined similarly to that of quantities. The evaluation for game boards, i. e., fluents with ordered input arguments but unordered output arguments, shall reflect the distance of the position of a piece on the board in the current state to the goal position of this piece.

The new evaluation function $eval(\texttt{true}(f), s)$ for fluents f that are identified as game boards, is identical to the evaluation function for quantities (Equation 5.9):

$$eval(\texttt{true}(f), s) = \begin{cases} p & \text{if } D \cup s^{\texttt{true}} \models \texttt{true}(f) \\ 0 & \text{if } \delta(s,f) = \infty \\ (1-t) * (1 - \delta(s,f)) & \text{otherwise} \end{cases}$$

However, the distance function $\delta(s, f)$ must be defined differently. For quantities, $\delta(s, f)$ is computed as the normalized difference between current and goal value of the quantity. The analogue for boards would be to use the normalized difference between current and goal location of a piece. However, there is a problem here: There are games with multiple pieces of the same name. For example, there are 8 pawns of each colour in Chess and 10 identical pieces for each player in Chinese checkers. Because we do not know which of these pieces shall be moved to the destination, we use the average distance of all matching pieces to the destination. For some games using the minimal or maximal distance of all matching pieces may be more appropriate. However, since this is game dependent we chose the average distance as a good compromise.

Thus, we define the distance function for game boards as follows. Let s be a game state and $f(x_1, \ldots, x_n, y_1, \ldots, y_m)$ be a ground instance of a fluent describing a game board. Without loss of generality, let

- $\{1, \ldots, n\}$ be the input arguments of f,
- $\{n+1, \ldots, n+m\}$ be the output arguments of f,
- $\vec{x} = (x_1, \ldots, x_n)$ be the values of the input arguments of the ground instance of f,
- $\vec{x'} = (x_1', \ldots, x_n')$ be an arbitrary vector of terms of length n,
- $\vec{y} = (y_1, \ldots, y_m)$ be the values of the output arguments of the ground instance of f,
- $s_f = \{f(\vec{x'},\vec{y}) \mid f(\vec{x'},\vec{y}) \in s\}$ be the set of all instances of $f(\vec{x'},\vec{y})$ in s, i. e., all fluents in s that are named f and have the same output arguments (content of the board cell) as $f(x_1, \ldots, x_n, y_1, \ldots, y_m)$ but arbitrary input arguments (coordinates of the board cell), and

- $r_1, \ldots, r_n$ be order relations for the domains of the respective input arguments of f.

We compute the distance from state s to $f(\vec{x}, \vec{y})$ as follows:

$$\delta(s, f(\vec{x}, \vec{y})) = \begin{cases} \frac{1}{|s_f|} * \sum_{f(\vec{x}', \vec{y}) \in s_f} \frac{1}{n} * \sum_i \frac{\Delta_{r_i}(x'_i, x_i)}{|dom(f,i)|} & \text{if } N > 0 \\ 1 & \text{otherwise} \end{cases} \tag{5.11}$$

where $|dom(f, i)|$ denotes the size of the domain of the i-th argument of f, and Δ_{r_i} is the difference function according to the order of the domain of the i-th input argument of f.

In the case that some of the input arguments of the board are not ordered, i. e., there are no order relations for their domains, we use the following simple difference function instead of the respective Δ_{r_i} in Equation 5.11:

$$\Delta_{\neg order}(a, b) = \begin{cases} 0 & \text{if } a = b \\ 1 & \text{otherwise} \end{cases}$$

As an example, consider the previously mentioned Racetrack game with the fluent `cell(B, Y, X, C)` denoting that the cell `X,Y` on board `B` contains `C`. This fluent is identified as a board with input arguments $\{1, 2, 3\}$ and output argument $\{4\}$, where the arguments 2 and 3 (i. e., `Y` and `X`) are ordered, but the argument 1 (`B`) is not. The goal condition for the white player contains, among other conditions, the atom `true(cell(wlane,e,1,white))`, that is, a white piece must be located on the board `wlane` in row `e` and column `1`. Suppose, we have the situation as shown in Figure 5.2: There is a white piece at location `b,2` on board `wlane` and no other white piece anywhere else. Thus, the distance is computed as

$$\begin{aligned} &\delta(s, \texttt{true}(\texttt{cell}(\texttt{wlane}, \texttt{e}, \texttt{1}, \texttt{white}))) \\ &= \frac{1}{3} * \left(\Delta_{\neg order}(\texttt{wlane}, \texttt{wlane}) + \frac{\Delta_{r_2}(\texttt{b}, \texttt{e})}{|dom(\texttt{cell}, 2)|} + \frac{\Delta_{r_3}(\texttt{2}, \texttt{1})}{|dom(\texttt{cell}, 3)|} \right) \\ &= \frac{1}{3} * \left(0 + \frac{3}{5} + \frac{1}{3} \right) = \frac{14}{45} \end{aligned}$$

We implemented the identification of structures and the use of these structures in the evaluation function as described in the previous sections in our general game player. In the next section, we will present an evaluation of the effectiveness of these improvements based on real games played against other general game playing programs in a competition.

5.4. Evaluation

It is difficult to find a fair evaluation scheme for general game playing systems because the performance of a general game player might depend on many different

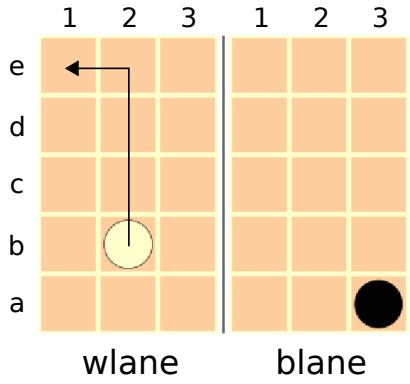

Figure 5.2.: An example state of the Racetrack game.

factors, e. g., the game played, the actual description of the game, the opponents and the time given for analysis of the game as well as for actual game play. In contrast to SAT solvers or planners, for general game players, there exists no predefined set of benchmark problems. So, instead of developing our own set of benchmark problems and running a lot of experiments with different parameters against random opponents, we chose to use the results of the international GGP competition for determining which parts of our evaluation function resulted in good play for which games.

The work presented so far in this chapter was implemented in the version of Fluxplayer that competed in the AAAI GGP Competition 2006. In this competition, more than 400 matches (more than 70 for each player) of different games were played. The advantage of using this data is that Fluxplayer was playing against real opponents. Thus, we can directly compare the performance of Fluxplayer to that of other state of the art systems.

We decided on the following evaluation scheme. First, we compared the score of Fluxplayer in different types of games to Fluxplayer's own average score in all games in order to find out in which games our evaluation function works best. The relative difference between Fluxplayer's score for each type of game to its average score in all games is shown in dark grey in Figure 5.3. Second, we compared Fluxplayer's score to that of the other players in each type of game in order to see in which games Fluxplayer has an advantage or disadvantage compared to other players. The relative difference between Fluxplayer's score and that of the other players is shown in light grey. Figure 5.3 depicts the performance of Fluxplayer separately for games with the following seven properties:

concrete The goal states of the game have a concrete set of properties. That is, the goal is a conjunction of ground fluents. One example is the 8-puzzle where the goal is to bring every piece of a sliding tile puzzle to a certain location. The goal condition is a conjunction of ground fluents describing the exact location of each of the 8 pieces.

gradual The game features gradual goal values. That is, the reward of the player is not only 100 or 0, but there are several steps in between in case only a part of the goal is reached. Games of the 2006 competition that have

this property include variants of Chinese checkers, where the goal value depends on the number of pieces in the goal area, variants of Checkers and Chess, where the goal value depends on the number of captured pieces.

quantity The goal involves a detected quantity or order relation. That is, the goal is to reach a certain amount of something. Examples are Othello as well as Blobwars. In both games the goal is to have more pieces than the opponent. In Othello, this is expressed by an order relation as described in Section 5.3.1 while in Blobwars there is a fluent `winning(Who,HowMany)` describing that player `Who` is winning by `HowMany` pieces. This fluent is recognized as a quantity by Fluxplayer.

distance The goal of the game involves a position of a piece on an ordered board such that a distance measure applies. Examples are the Racetrack game described above and Chinese checkers, among others.

large b The game has a *large* branching factor. That is, in at least one of the states that occurred during a match one of the players had more than 20 legal moves. Games with a large branching factor can only be searched to a very small depth. Thus, heuristic search typically does not yield good results for such games.

nondescript The goal condition of the game is nondescript, i. e., contains little or no information that distinguishes between sibling states in the game tree. For example, the game Peg has the goal condition `true(pegs(0))`, indicating that the game is won if there is no peg left. However, the game is constructed in such a way that one peg is removed from the board in each step. That is, all states in the same depth of the game tree have the same number of pegs and can thus not be distinguished by an evaluation function that is solely based on the goal condition. The actual (implicitly defined) goal of the game is to avoid terminal states until all pegs are removed.

mobility The goal or terminal conditions of the game amount to a *mobility* heuristics, which is the case, e. g., for games ending if there are no more legal moves. The intuition behind the mobility heuristics is that a state in which a player has more legal moves tends be be better for this player because then this player has more control over the course of the game.

We chose the properties above because either our evaluation function is geared towards games with those properties (e. g., concrete, gradual, quantity, distance, mobility), or because we expect our player to have problems with those games (e. g., nondescript, large b). Of course, these properties do not define strict categories of games. That is, there are games that have more than one of the properties. The results of those games are than included in all of the respective categories in Figure 5.3.

From the results shown in Figure 5.3 we can draw the following conclusions for the separate classes of games:

concrete, gradual We see that Fluxplayer performs better that its own average in games with concrete goals and gradual goals. If a game has a concrete goal then Fluxplayer prefers those non-terminal states in which more of the fluents of the conjunction hold because of our particular evaluation

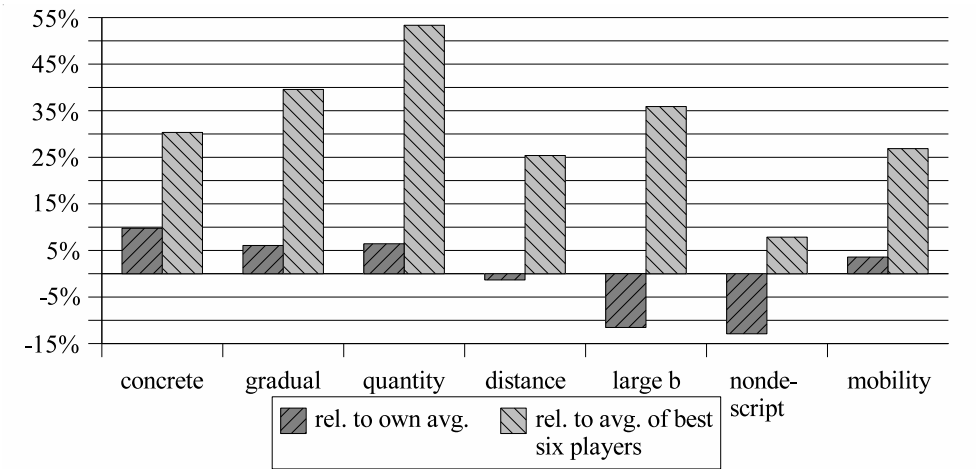

Figure 5.3.: The chart shows the average score of Fluxplayer for certain games relative to (1) the overall average score of Fluxplayer in all games and (2) the average score of the best six players for the games with the respective property.

function. Our evaluation function also takes advantage of gradual goals by weighting the evaluation of each distinct goal condition with the associated goal value (cf. Definition 5.3). The good performance in games with concrete goals and gradual goals indicates that our particular construction of the evaluation function is effective for those games.

quantity The same argument holds for games with goals involving quantities. Fluxplayer obtains more points in these games than on average. This indicates that our detection of quantities has a positive influence on the performance. We can also see that Fluxplayer performs much better than all the opponents (over 50% higher score), for games involving quantities. This indicates that the other players do not take as much advantage of this information as Fluxplayer does.

distance On first sight, it seems that the evaluation function for game boards is less effective than the one for quantities. After all, Fluxplayer's score in games involving a distance are even slightly below its own average. However, the results look worse than they are. The reason for the lower scores in these games is that many of the games with the distance property were so simple that all players could play them well. Furthermore, many of the games were zero-sum games and ended in a draw. Thus, the score of Fluxplayer in those games compared to its own average in all games is lower because the games were essentially played against perfect opponents. Fluxplayer still played the few non-trivial games in this class better than its opponents.

large b For games with a high branching factor, the scores of Fluxplayer are below its own average. This could be expected. After all, the search depth that Fluxplayer reaches in those games is much smaller. Still, Fluxplayer seems to handle these games much better than its opponents, which,

can have two reasons. First, due to the non-uniform depth search (see Section 4.4), Fluxplayer reaches a higher maximal depth during search. Furthermore, some of the games with a high branching factor also have other properties that Fluxplayer takes advantage of, such as, gradual goals, quantities or distances.

nondescript The main disadvantage of our approach is the dependency on the goal description. This is indicated by the poor performance for games with nondescript goals. Fluxplayer has still a slight advantage over its opponents, which may be caused by the fact that some of the games with the nondescript property also fall into the mobility category.

mobility Mobility seems to be a good heuristics in a wide range of games. Although not explicitly built into Fluxplayer, our evaluation function is similar to a mobility heuristics, if the goal or terminal condition of the game states that the game ends and is lost if a player has no legal move in a state.

5.5. Summary

In this chapter, we presented a method to evaluate states against a logic description of a goal with the help of fuzzy logics. We showed how to improve out evaluation function by acquiring and using knowledge about the game, such as game boards and quantities.

Although the terminology we use suggests that we know the meaning of the detected structures, we have no way of knowing what the game designer had in mind when he encoded the game. Thus, terms like "game board", "quantity" and "piece" are used for illustration and because of the lack of a better terminology. The actual concepts are much more abstract. The evaluation function for game boards works if the game has no board at all and there are no pieces. Finding input arguments of the fluents and order relations for the domains of the arguments of the fluents is all that is necessary. For example, a state in the blocks world domain is typically described using a set of fluents of the form `on(Block1,Block2)` with the intended meaning that `Block1` is lying directly on top of `Block2`. Because of the nature of the domain, a block can only lie on at most one other block. Thus, the first argument of the fluent `on` is an input argument and `on` is recognized as a one-dimensional board.

Furthermore, we presented an evaluation of the performance of Fluxplayer in games that exhibit certain properties. Our evaluation has shown that our state evaluation function and the improvements based on detected structures are effective. However, the evaluation function is dependent on the game rules. This may result in suboptimal performance if the goal condition of a game contains little or no information that distinguishes between states that are siblings in the game tree.

6. Distance Estimates for Fluents and States

In the previous chapter, we presented our state evaluation function and a way to improve it by identifying structures in the game. We use the obtained information for computing some form of difference between quantities or distances on the board, respectively. In particular, we defined a function $\delta(s, f)$, that estimates the distance between the state s and an arbitrary state that contains the fluent f. We defined this distance function in terms of order relations for the domains of the arguments of f. Although this approach often yields good results, the distance function obtained in this way has no connection to the way in which the fluents change from one state to the next. This may lead to suboptimal or misleading distance estimates in some games as we will show with an example in the next section.

In this chapter, we present a method to extract information about the possible changes of fluents from the `next` and `legal` rules of a game. Furthermore, we use this information to define a new distance estimate $\delta(s, f)$ in order to improve the state evaluation function presented in the previous chapter.

This chapter originates from joint work with my colleague and fellow PhD student Daniel Michulke.

6.1. Motivating Example

Before we define our new distance estimates, we will present a motivating example. Consider a game played on a chess board with the standard chess pieces. In a typical game description, the positions of pieces on the board are encoded using fluents such as `cell(X, Y, Piece)`, where `X` and `Y` refer to the coordinates of the location of the piece `Piece` on the board. With the method described in Section 5.2, we identify `cell` as a board, where the first two arguments (`X`, `Y`) are ordered coordinates (input arguments) and the third argument is an unordered output argument. Suppose the goal of the game contains the condition `cell(4,8,white_knight)`, that means that a white knight is standing at location `4,8`. In this case, we compute, as a part of our evaluation function, the distance $\delta(s, \texttt{cell(4,8,white\_knight)})$ of the current state s to an arbitrary state that contains the fluent `cell(4,8,white_knight)`. As defined in Section 5.3, the distance function δ computes the average distance of all white knights on the board to location `4,8` roughly according to the Manhattan distance, also called city-block metrics. In Chess, the Manhattan distance between two locations on the board is exactly the number of moves

it takes to move a king from one location to the other (ignoring all the other pieces and threats on the board). However, the Manhattan distance is not a good estimate for the number of moves needed by other pieces to cover the same distance. For example, it takes three moves with a knight to reach an adjacent cell of the board, while the Manhattan distance is only 1 in this case. An extreme example is the movement of bishops. Bishops, can never be moved to a square with a different color. Thus, they can never reach a square adjacent to their current location. In this case, the distance between adjacent locations should be infinite as opposed to 1, which is the Manhattan distance.

These examples show that the distance function used in Section 5.3 can lead to suboptimal estimates of the real distances. The reason is that the distance function is only based on the structure of the board but not on the way the contents of the board cells change, i. e., the way pieces move.

The idea for the approach presented in this chapter is to analyse the `next` and `legal` rules of the game to find potential ways in which fluents change from one state to the next. This information is then be used to define a new distance estimate $\delta(s, f)$, which can be plugged in the state evaluation function presented in Chapter 5. Applied to the example above, this means that we can detect the move patterns for every type of piece and compute a distance based on the move pattern. However, the approach is not limited to board games. Distances can be computed for all kinds of fluents.

6.2. Distance Estimates Derived from Fluent Graphs

We analyse the game in order to obtain a distance estimate for the atomic subgoals of the game, i. e., the fluents. More precisely, we want to compute an estimate of how many steps are needed to make a certain fluent true given the current state of the game. We start by building a *fluent graph* that contains all the fluents of a game as nodes in the following section. Then we define a distance function based on a fluent graph in Section 6.2.2.

6.2.1. Fluent Graphs

The fluent graph has directed edges (f_i, f) between the fluents such that at least one of the predecessor fluents f_i must hold in the current state if fluent f shall hold in the successor state.

Figure 6.1 shows a partial fluent graph for Tic-Tac-Toe (see Figure 2.1 for the rules) that relates the fluents `cell(c,1,Z)` for `Z` $\in$ `{b, x, o}`. One can see that for cell `(c,1)` to be blank it had to be blank before. For the cell to contain an `x` (or an `o`) in the successor state, there are two possible preconditions. Either, it contained an `x` (or `o`) before or it was blank.

By design, a fluent graph is an abstraction of a game. It contains only some of the necessary preconditions for a fluent to hold in the successor state. However, in general, it does not contain all of the necessary preconditions. For example,

if cell `(c,1)` was blank in the current state it also had to be marked by the xplayer to contain an `x` in the successor state. This would require for xplayer to be in control, that is, `control(xplayer)` would have to hold. However, this precondition is not contained in the graph because the graph is an abstraction of the game. Nevertheless, the graph contains valuable information. For example, one can see that `cell(c,1,x)` can not be fulfilled if `cell(c,1,blank)` does not hold (e. g., if the cell is marked by the `oplayer`) because the only path to `cell(c,1,x)` starts at `cell(c,1,blank)`.

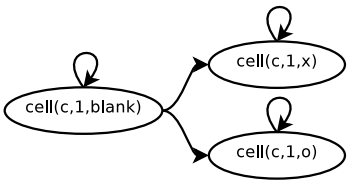

Figure 6.1.: Partial fluent graph for Tic-Tac-Toe showing the dependencies between the different contents of a cell.

Formally, a fluent graph is defined as follows:

Definition 6.1 (Fluent Graph)**.** *Let* $\Gamma = (R, s_0, T, l, u, g)$ *be a game with ground terms* Σ*. A graph* $G = (V, E)$ *is called a fluent graph for* Γ *iff*

- $V \subseteq \Sigma \cup \{\emptyset\}$ *and*
- *for all fluents* f'*, reachable states* s *and joint actions* A*:*

$$\neg T(s) \wedge l(A, s) \wedge f' \in u(A, s) \supset (\exists f)(f, f') \in E \wedge f \in s \cup \{\emptyset\}$$

Thus, the nodes of the graph are all fluents of the game and the additional node $\emptyset$. Let us ignore this additional node for a moment. Then, the edges of the graph are defined in the following way. We consider every state transition $s \mapsto u(A, s)$ of the game, where s is a reachable non-terminal state of the game and A is a legal joint move in s. For each fluent f' that occurs in the successor state $u(A, s)$ we require the fluent graph to contain an edge from at least one fluent $f \in s$ to f'. The fluent graph of a game can be intuitively understood as an abstraction of the game:

- A node f in the fluent graph corresponds to the set of all states that contain the fluent f.
- An edge (f, f') in the fluent graph corresponds to the set of all state transitions from a state that contain the fluent f to a state that contains f'.

Intuitively, the length of a path from f to f' in the fluent graph is, therefore, an approximation of the number of state transitions, i. e. steps, needed to go from a state containing f to a state containing f'.

So, why do we need the special node $\emptyset$ in Definition 6.1? We motivated the edges in the fluent graph leading to a fluent f' as preconditions of f'. However, there can be fluents in the game that do not have any preconditions. For example, the fluent g with the next rule `next(g) :- distinct(a,b).` has no fluent as a precondition. On the other hand, a fluent h with the next rule `next(h) :- distinct(a,a).` cannot occur in any successor state because the body of its next rule is unsatisfiable. We want to distinguish between fluents that have no precondition (such as g) and fluents that will never be reached (such as h), i. e., where no successor state $u(A, s)$ exists such that $h \in u(A, s)$. Therefore, if there is no precondition for a fluent g, g is connected to the node $\emptyset$, while a fluent h that cannot be reached, will have no edge in the fluent graph.

As stated before, a fluent graph contains only some of the necessary preconditions for fluents. When constructing a fluent graph, one can choose which preconditions of a fluent to add as edges to the fluent graph. Thus, fluent graphs for a game are not unique. For example, Figure 6.2 shows an alternative partial fluent graph for Tic-Tac-Toe that also contains the fluents `cell(c,1,Z)` for `Z` $\in$ `{b, x, o}`. We will address the implications of this property of fluent graphs in Section 6.3.2.

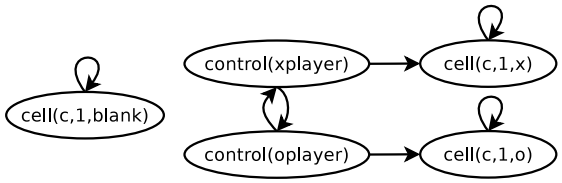

Figure 6.2.: Alternative partial fluent graph for Tic-Tac-Toe showing the dependencies between the different contents of cell `(c,1)` and whose turn it is.

6.2.2. Distance Estimates

We can now define a distance function $\Delta(s, f')$ between the current state s and a state in which fluent f' holds as follows:

Definition 6.2 (Distance Function $\Delta(s, f')$)**.** *Let $\Gamma = (R, s_0, T, l, u, g)$ be a game with ground terms Σ and $G = (V, E)$ be a fluent graph for Γ. Furthermore, let $\Delta_G(f, f')$ be the length of the shortest path from node f to node f' in the fluent graph G or ∞ if there is no such path. Then*

$$\Delta(s, f') = \min_{f \in s \cup \{\emptyset\}} \Delta_G(f, f')$$

That means, we compute the distance $\Delta(s, f')$ as the shortest path in the fluent graph from any fluent in s to f'. Intuitively, each edge (f, f') in the fluent graph corresponds to a state transition of the game from a state in which f holds to a state in which f' holds. Thus, the length of a path from f to f' in the

fluent graph corresponds to the number of steps in the game between a state containing f to a state containing f'.

Of course, the fluent graph is an abstraction of the actual game: many preconditions for the state transitions are ignored. As a consequence, the distance $\Delta(s, f')$ that we compute in this way is always a lower bound on the actual number of steps it takes to go from s to a state in which f' holds. Therefore, the distance $\Delta(s, f')$ is an admissible heuristics for the number of steps it takes to fulfil f' starting in state s. We state this in the following theorem.

Theorem 6.1 (Admissible Distance)**.** *Let*

- $\Gamma = (R, s_0, T, l, u, g)$ *be a game with ground terms* Σ *and states* $\mathcal{S}$*,*
- $s_1 \in \mathcal{S}$ *be a state of* Γ*,*
- $f \in \Sigma$ *be a fluent of* Γ*, and*
- $G = (V, E)$ *be a fluent graph for* Γ*.*

Furthermore, let $s_1 \mapsto s_2 \mapsto \ldots \mapsto s_{m+1}$ *denote a legal sequence of states of* Γ*, that is, for all* i *with* $0 < i \leq m$ *there is a joint action* A_i*, such that:*

$$s_{i+1} = u(A_i, s_i) \wedge l(A_i, s_i)$$

If $\Delta(s_1, f) = n$*, then there is no legal sequence of states* $s_1 \mapsto \ldots \mapsto s_{m+1}$ *with* $f \in s_{m+1}$ *and* $m < n$*.*

Proof. We prove the theorem by contradiction. Assume that $\Delta(s_1, f) = n$ and there is a a legal sequence of states $s_1 \mapsto \ldots \mapsto s_{m+1}$ with $f \in s_{m+1}$ and $m < n$. By Definition 6.1, for every two consecutive states s_i, s_{i+1} of the sequence $s_1 \mapsto \ldots \mapsto s_{m+1}$ and for every $f_{i+1} \in s_{i+1}$ there is an edge $(f_i, f_{i+1}) \in E$ such that $f_i \in s_i$ or $f_i = \emptyset$. Therefore, there is a path $f_j, \ldots, f_m, f_{m+1}$ in G with $1 \leq j \leq m$ and the following properties:

- $f_i \in s_i$ for all $i = j, ..., m + 1$,
- $f_{m+1} = f$, and
- either $f_j \in s_1$ (e. g., if $j = 1$) or $f_j = \emptyset$.

Thus, the path $f_j, \ldots, f_m, f_{m+1}$ has a length of at most m. Consequently, $\Delta(s_1, f) \leq m$ because $f_j \in s_1 \cup \{\emptyset\}$ and $f_{m+1} = f$. However, $\Delta(s_1, f) \leq m$ together with $m < n$ contradicts $\Delta(s_1, f) = n$. □

In the following, we explain how the distance function $\Delta(s, f')$ (Definition 6.2) can be used in our state evaluation function.

In our evaluation function as defined in Chapter 5 we use distance estimates $\delta(s, f)$ that output values in the range $[0, 1]$ for boards and quantities in the following way (see Section 5.3.2):

$$eval(\texttt{true}(f), s) = \begin{cases} p & \text{if } D \cup s^{\texttt{true}} \models \texttt{true}(f) \\ 0 & \text{if } \delta(s, f) = \infty \\ (1 - t) * (1 - \delta(s, f)) & \text{otherwise} \end{cases} \tag{6.1}$$

With the distance estimates developed in this chapter, we can use the same evaluation function for arbitrary fluents instead of only those that were identified as boards or quantities. However, we still need to define the normalized distance $\delta(s, f)$. For this, we use our distance function $\Delta(s, f)$ from Definition 6.2 as follows:

$$\delta(s, f) = \frac{\Delta(s, f)}{\Delta_{max}(f) + 1} \tag{6.2}$$

The value $\Delta_{max}(f)$ is the longest distance $\Delta_G(g, f)$ from any fluent g to f, i. e.,

$$\Delta_{max}(f) \stackrel{\text{def}}{=} \max_g \Delta_G(g, f)$$

Thus, $\Delta_{max}(f)$ is the longest possible distance $\Delta(s, f)$ that is not infinite. Note that $\delta(s, f)$ will be infinite if $\Delta(s, f) = \infty$, i. e., if there is no path from any fluent in s to f in the fluent graph. In all other cases $0 \leq \delta(s, f) < 1$. The denominator in Equation 6.2 is $\Delta_{max}(f) + 1$ to ensure that $\delta(s, f) < 1$ (as opposed to ≤ 1) if there is a path to f. If we would allow $\delta(s, f) = 1$, the evaluation function (Equation 6.1) could not distinguish between the two cases that

1. $\delta(s, f) = \infty$, i. e., f is not reachable and
2. $\delta(s, f) = 1$, i. e., f is reachable but the distance is maximal,

because, in both cases, $eval(\texttt{true}(f), s) = 0$.

6.3. Constructing Fluent Graphs from Rules

The definition of our distance function $\delta(s, f)$ refers to paths in a fluent graph of the game. Thus, to compute the distance, we need to construct a fluent graph, first. However, the definition of fluent graphs (Definition 6.1) is not directly suited for actually constructing a fluent graph for a game. Even in games of low complexity, considering all state transitions is hardly feasible. Therefore, we propose an algorithm to construct a fluent graph based on the rules of the game as opposed to the state transitions of the game.

We motivate our algorithm as follows: Intuitively, an edge (f, f') in the fluent graph of a game indicates that f is a precondition of f', that is, for f' to hold in a successor state, f has to hold in one of the predecessor states. The conditions when a fluent f' holds in a successor state are defined by the `next` rules of the game description. More precisely, if f' holds in the successor state than `next(f')` must hold in the current state. Thus, if the body of a `next` rule for f' contains an atom `true(f)` (which is true if f holds in the current state), than f is a precondition of f' and, thus, a candidate for the source of an edge (f, f') in the fluent graph.

As stated before, the definition of a fluent graph allows for several different fluent graphs of game. When constructing a fluent graph the goal should be to obtain a graph that gives rise to a distance function (cf. Definition 6.2) that is as accurate as possible. Since the distance function is admissible regardless of which fluent graph is used as a basis, we should construct a fluent graph

that maximises the distances between its nodes. Therefore, we only add edges in the graph if necessary because in a graph with less edges the average path length between two nodes (and thus the distance) is greater. Furthermore, we try to avoid adding edges from the special node $\emptyset$ because the existence of the edge $(\emptyset, f')$ limits the maximal distance $\Delta(s, f')$ from any state to f' to 1 (cf. Definition 6.2).

With this guidelines in mind, we propose Algorithm 2 to construct a fluent graph from the rules of the game. The algorithm constructs a ground formula ϕ in disjunctive normal form that is an abstraction of $\texttt{next}(f')$, i. e., $\texttt{next}(f') \supset \phi$ (cf. line 4). Without loss of generality, we can assume that $\phi = \psi_1 \vee \ldots \vee \psi_n$, and every $\psi_i = l_{i,1} \wedge \ldots \wedge l_{i,m_i}$, where the $l_{i,j}$ are (possibly negated) ground atoms. The formula $\texttt{next}(f')$ holds in all predecessor states of state transitions leading to a state in which f' holds. Thus, every disjunct ψ_i of ϕ holds in a subset of these predecessor states. Hence, if there is a $l_{i,j} = \texttt{true}(f)$ in ψ_i, the fluent f also holds in all these predecessor states and f is potential candidate for the source of an edge to f'. In line 8, we select one such f for every ψ_i. We select only one, because our first guideline is to minimize the number of edges in the graph. Only if there is no literal of the form $\texttt{true}(f)$ in ψ_i, do we add the edge $(\emptyset, f')$.

Algorithm 2 Constructing a fluent graph from a game description.

Input: game description D, set of all ground fluents F of the game
Output: fluent graph $G = (V, E)$

```
1:  V := ∅, E := ∅
2:  for all f' ∈ F do
3:      V := V ∪ {f'}
4:      Construct a ground formula φ in disjunctive normal form, such that
        next(f') ⊃ φ.
5:      Let φ = ψ_1 ∨ ... ∨ ψ_n.
6:      for all i ∈ {1, ..., n} do
7:          Let ψ_i = l_{i,1} ∧ ... ∧ l_{i,m_i}.
8:          Select a fluent f such that (∃j) l_{i,j} = true(f)
9:          if there is no such fluent then
10:             f := ∅
11:         end if
12:         V := V ∪ {f}, E := E ∪ {(f, f')}
13:     end for
14: end for
```

The following theorem states that the graph constructed by Algorithm 2, is a valid fluent graph of the game.

Theorem 6.2 (Soundness of the Fluent Graph Construction)**.** *Let D be a valid GDL game specification and Γ be the associated game. Let $G = (V, E)$ be the graph constructed by running Algorithm 2 with the game description D. Then G is a fluent graph of Γ according to Definition 6.1.*

Proof. Let $\Gamma = (R, s_0, T, l, u, g)$ and Σ be the ground terms of the game. The nodes V of the graph are a subset of $\Sigma \cup \{\emptyset\}$ because the algorithm only adds fluents of the game and $\emptyset$ as nodes and all fluents are terms from Σ.

Assume that the condition on E in Definition 6.1 is not fulfilled, that is, there is a state s, a joint action A that is legal in s, and a fluent $f' \in u(A, s)$ such that there is no edge $(f, f') \in E$ for any $f \in s \cup \{\emptyset\}$.

If there is a fluent f' in the state $u(A, s)$ then, by Definition 2.14, $D \cup s^{\texttt{true}} \cup A^{\texttt{does}} \models \texttt{next}(f')$. The formula $\phi = \psi_1 \vee \ldots \vee \psi_n$ in Algorithm 2 is implied by $\texttt{next}(f')$. Thus, there must be at least one $i \in \{1, \ldots, n\}$ such that $D \cup s^{\texttt{true}} \cup A^{\texttt{does}} \models \psi_i$. Now, we make a case distinction:

1. Let ψ_i contain a positive literal of the form $\texttt{true}(f)$. Without loss of generality, let f be the fluent selected by Algorithm 2 in line 8. Then, the algorithm adds the edge (f, f'). Furthermore, $f \in s$ because $D \cup s^{\texttt{true}} \cup A^{\texttt{does}} \models \psi_i$ implies $D \cup s^{\texttt{true}} \cup A^{\texttt{does}} \models \texttt{true}(f)$. This contradicts our assumption that there is no edge $(f, f') \in E$ for any $f \in s \cup \{\emptyset\}$.
2. Let ψ_i contain no literal of the form $\texttt{true}(f)$. Then, the algorithm adds the edge $(\emptyset, f')$, which also contradicts our assumption.

Thus, the theorem is proved by contradiction. □

The algorithm outline still leaves some open issues:

1. How do we construct the ground formula ϕ for $\texttt{next}(f')$ (line 4)?
2. Which literal $\texttt{true}(f)$ do we select from a conjunction ψ_i if there is more than one (line 8)? Or, in other words, which precondition f of f' do we select if there are several preconditions?

We will discuss both issues in the following sections.

6.3.1. Constructing ϕ

Given a ground fluent f', we want to construct a formula ϕ in disjunctive normal form (DNF) that is implied by $\texttt{next}(f')$. A formula ϕ is in DNF if $\phi = \psi_1 \vee \ldots \vee \psi_n$ and each formula ψ_i is a conjunction of literals $\psi_i = l_{i,1} \wedge \ldots \wedge l_{i,m_i}$. Thus, the formula $\texttt{next}(f')$ itself is already in disjunctive normal form because it is an atom. Although using $\phi = \texttt{next}(f')$ in Algorithm 2 would result in a valid fluent graph, the resulting fluent graph would be trivial. Since $\texttt{next}(f')$ does not contain any literals of the form $true(f)$, the fluent graph would only contain edges of the form $(\emptyset, f')$. Thus, all distances obtained from the fluent graph would be at most 1. Hence, when constructing the formula ϕ, we must strive for a formula that is informative, in the sense that it contains the most relevant preconditions of f'.

Therefore, we propose Algorithm 3 to construct ϕ. The algorithm starts with $\phi = \texttt{next}(f')$. Then, it selects a positive literal l in ϕ and unrolls this literal, that is, it replaces l with the bodies of all rules whose head matches l. This replacement is done in lines 8 and 9. The replacement is repeated until all predicates that are left are either `true`, `distinct`, or `does` or recursively defined.

Algorithm 3 Constructing a formula ϕ in DNF with $\phi \equiv \texttt{next}(f')$.

Input: game description D, ground fluent f'
Output: ϕ, such that $\phi \equiv \texttt{next}(f')$

1: $\phi := \texttt{next}(f')$
2: $finished := false$
3: **while** $\neg finished$ **do**
4: Select a positive literal l from ϕ such that $l \neq \texttt{true}(t), l \neq \texttt{distinct}(t_1, t_2), l \neq \texttt{does}(r, a)$ and there is no cycle including l in the dependency graph of D.
5: **if** there is no such literal **then**
6: $finished := true$
7: **else**
8: $\hat{l} := \bigvee_{h:-b \in D, l\sigma = h\sigma} b\sigma$
9: $\phi := \phi\{l/\hat{l}\}$
10: **end if**
11: **end while**
12: Transform ϕ into disjunctive normal form, i. e., $\phi = \psi_1 \vee \ldots \vee \psi_n$ and each formula ψ_i is a conjunction of literals.
13: **for all** ψ_i in ϕ **do**
14: Replace ψ_i in ϕ by a disjunction of all ground instances of ψ_i.
15: **end for**

Recursively defined predicates are not unrolled to ensure termination of the algorithm. Finally, we transform ϕ into disjunctive normal form and replace each disjunct ψ_i of ϕ by a disjunction of all of its ground instances in order to get a ground formula ϕ. The ground instances of a formula can be easily computed using the domain information that is obtained as presented in Section 5.2.1.

Note that we do not select negative literals for unrolling. The algorithm could be easily adapted to also unroll negative literals. However, in the games we encountered so far, doing so does not improve the obtained fluent graphs but complicates the algorithm and increases the size of the created ϕ. Unrolling negative literals will mainly add negative preconditions to ϕ. However, negative preconditions are not used for the fluent graph because a fluent graph only contains positive preconditions of fluents as edges, according to Definition 6.1.

We present the following example to illustrate how Algorithm 3 works. Consider the following rules of Tic-Tac-Toe:

```
1 next(cell(M,N,x)) :- does(xplayer,mark(M,N)).
2 next(cell(M,N,o)) :- does(oplayer,mark(M,N)).
3 next(cell(M,N,C)) :- true(cell(M,N,C)),
4     does(P,mark(X,Y)), distinct(X,M).
5 next(cell(M,N,C)) :- true(cell(M,N,C)),
6     does(P,mark(X,Y)), distinct(Y,N).
```

The formula ϕ for `next(cell(c,1,x))` can be easily constructed by taking a

disjunction of the bodies of all but the second rule:

$$\begin{aligned}\phi \;=\;& \texttt{does}(\texttt{xplayer}, \texttt{mark}(\texttt{c}, \texttt{1})) \vee \\ & \texttt{true}(\texttt{cell}(\texttt{c}, \texttt{1}, \texttt{x})) \wedge \texttt{does}(\texttt{P}, \texttt{mark}(\texttt{X}, \texttt{Y})) \wedge \texttt{distinct}(\texttt{X}, \texttt{c}) \vee \\ & \texttt{true}(\texttt{cell}(\texttt{c}, \texttt{1}, \texttt{x})) \wedge \texttt{does}(\texttt{P}, \texttt{mark}(\texttt{X}, \texttt{Y})) \wedge \texttt{distinct}(\texttt{Y}, \texttt{1})\end{aligned}$$

Thus, only one iteration of the loop in line 3 is necessary. Finally, to compute all ground instances, we use the domain information that we obtained for the game (see Section 5.2.1). We know that `X` (the first argument of the `mark` move) ranges over $\{\texttt{a}, \texttt{b}, \texttt{c}\}$, `Y` ranges over $\{\texttt{1}, \texttt{2}, \texttt{3}\}$, and `P` ranges over the players $\{\texttt{xplayer}, \texttt{oplayer}\}$. Thus, we obtain the ground form of ϕ as depicted in Figure 6.3.

$$\begin{aligned}\phi \;=\;& \texttt{does}(\texttt{xplayer}, \texttt{mark}(\texttt{c}, \texttt{1})) \vee \\ & \texttt{true}(\texttt{cell}(\texttt{c}, \texttt{1}, \texttt{x})) \wedge \texttt{does}(\texttt{xplayer}, \texttt{mark}(\texttt{a}, \texttt{1})) \wedge \texttt{distinct}(\texttt{a}, \texttt{c}) \vee \\ & \texttt{true}(\texttt{cell}(\texttt{c}, \texttt{1}, \texttt{x})) \wedge \texttt{does}(\texttt{xplayer}, \texttt{mark}(\texttt{b}, \texttt{1})) \wedge \texttt{distinct}(\texttt{b}, \texttt{c}) \vee \\ & \ldots \vee \\ & \texttt{true}(\texttt{cell}(\texttt{c}, \texttt{1}, \texttt{x})) \wedge \texttt{does}(\texttt{xplayer}, \texttt{mark}(\texttt{c}, \texttt{3})) \wedge \texttt{distinct}(\texttt{c}, \texttt{c}) \vee \\ & \texttt{true}(\texttt{cell}(\texttt{c}, \texttt{1}, \texttt{x})) \wedge \texttt{does}(\texttt{xplayer}, \texttt{mark}(\texttt{a}, \texttt{1})) \wedge \texttt{distinct}(\texttt{1}, \texttt{1}) \vee \\ & \ldots \vee \\ & \texttt{true}(\texttt{cell}(\texttt{c}, \texttt{1}, \texttt{x})) \wedge \texttt{does}(\texttt{xplayer}, \texttt{mark}(\texttt{c}, \texttt{3})) \wedge \texttt{distinct}(\texttt{3}, \texttt{1}) \vee \\ & \texttt{true}(\texttt{cell}(\texttt{c}, \texttt{1}, \texttt{x})) \wedge \texttt{does}(\texttt{oplayer}, \texttt{mark}(\texttt{a}, \texttt{1})) \wedge \texttt{distinct}(\texttt{a}, \texttt{c}) \vee \\ & \ldots \vee \\ & \texttt{true}(\texttt{cell}(\texttt{c}, \texttt{1}, \texttt{x})) \wedge \texttt{does}(\texttt{oplayer}, \texttt{mark}(\texttt{c}, \texttt{3})) \wedge \texttt{distinct}(\texttt{c}, \texttt{c}) \vee \\ & \texttt{true}(\texttt{cell}(\texttt{c}, \texttt{1}, \texttt{x})) \wedge \texttt{does}(\texttt{oplayer}, \texttt{mark}(\texttt{a}, \texttt{1})) \wedge \texttt{distinct}(\texttt{1}, \texttt{1}) \vee \\ & \ldots \vee \\ & \texttt{true}(\texttt{cell}(\texttt{c}, \texttt{1}, \texttt{x})) \wedge \texttt{does}(\texttt{oplayer}, \texttt{mark}(\texttt{c}, \texttt{3})) \wedge \texttt{distinct}(\texttt{3}, \texttt{1})\end{aligned}$$

Figure 6.3.: The ground formula ϕ in DNF for **`next(cell(a,1,x))`**.

The formula ϕ in Figure 6.3 can now be used to construct a part of the fluent graph for Tic-Tac-Toe according to Algorithm 2. Of course, to construct the complete fluent graph, we have to consider the formula ϕ for every fluent f' of the game. We can see that all but the first disjunct of ϕ in Figure 6.3 contain the literal $\texttt{true}(\texttt{cell}(\texttt{c}, \texttt{1}, \texttt{x}))$. Thus, for those disjuncts, Algorithm 2 could add the edge $(\texttt{cell}(\texttt{c}, \texttt{1}, \texttt{x}), \texttt{cell}(\texttt{c}, \texttt{1}, \texttt{x}))$ to the fluent graph. The first disjunct does not contain a literal of the form $\texttt{true}(f)$. Thus, we have to add the edge $(\emptyset, \texttt{cell}(\texttt{c}, \texttt{1}, \texttt{x}))$. However, adding the edge $(\emptyset, \texttt{cell}(\texttt{c}, \texttt{1}, \texttt{x}))$ is undesirable. According to the definition of the distance function (Definition 6.2), the distance $\Delta(s, f)$ is at most 1 if there is an edge $(\emptyset, f)$ in the fluent graph[1]. Thus, we want

[1]Remember, that our goal is to maximize $\Delta(s, f)$ because it is admissible.

to avoid using the $\emptyset$ node in the fluent graph, if possible. We can accomplish that in many games by a small modification of Algorithm 3.

The new algorithm (Algorithm 4) differs from Algorithm 3 only by the addition of line 4. As a consequence the resulting formula ϕ is no longer equivalent to $\texttt{next}(f')$. However, $\texttt{next}(f') \supset \phi$, under the assumption that only legal moves can be executed, i. e., $\texttt{does}(r, a) \supset \texttt{legal}(r, a)$. This is sufficient for constructing a fluent graph from ϕ according to Algorithm 2, where Algorithm 4 is used.

Algorithm 4 Constructing a formula ϕ in DNF with $\texttt{next}(f') \supset \phi$.

Input: game description D, ground fluent f'
Output: ϕ, such that $\texttt{next}(f') \supset \phi$

1: $\phi := \texttt{next}(f')$
2: $finished := false$
3: **while** $\neg finished$ **do**
4: Replace every positive occurrence of $\texttt{does}(r, a)$ in ϕ with $\texttt{legal}(r, a)$.
5: Select a positive literal l from ϕ such that $l \neq \texttt{true}(t), l \neq \texttt{distinct}(t_1, t_2), l \neq \texttt{does}(r, a)$ and there is no cycle including l in the dependency graph of D.
6: **if** there is no such literal **then**
7: $finished := true$
8: **else**
9: $\hat{l} := \bigvee_{h:-b \in D, l\sigma = h\sigma} b\sigma$
10: $\phi := \phi\{l/\hat{l}\}$
11: **end if**
12: **end while**
13: Transform ϕ into disjunctive normal form, i. e., $\phi = \psi_1 \vee \ldots \vee \psi_n$ and each formula ψ_i is a conjunction of literals.
14: **for all** ψ_i in ϕ **do**
15: Replace ψ_i in ϕ by a disjunction of all ground instances of ψ_i.
16: **end for**

Let us review the Tic-Tac-Toe example considering this change to the algorithm. Consider these additional rules of Tic-Tac-Toe:

```
1 legal(P, mark(X,Y)) :-
2     true(control(P)), true(cell(X,Y,blank)).
3 legal(P,noop) :-
4     role(P), not true(control(P)).
```

With Algorithm 4, we obtain the following formula ϕ for `next(cell(c,1,x))`

before the grounding step:

$$
\begin{aligned}
\phi \;=\; & \texttt{true}(\texttt{control}(\texttt{xplayer})) \wedge \texttt{true}(\texttt{cell}(\texttt{c},\texttt{1},\texttt{blank})) \\
& \vee \\
& \texttt{true}(\texttt{cell}(\texttt{c},\texttt{1},\texttt{x})) \wedge \texttt{true}(\texttt{control}(\texttt{P})) \wedge \texttt{true}(\texttt{cell}(\texttt{X},\texttt{Y},\texttt{blank})) \\
& \quad \wedge \texttt{distinct}(\texttt{X},\texttt{c}) \\
& \vee \\
& \texttt{true}(\texttt{cell}(\texttt{c},\texttt{1},\texttt{x})) \wedge \texttt{true}(\texttt{control}(\texttt{P})) \wedge \texttt{true}(\texttt{cell}(\texttt{X},\texttt{Y},\texttt{blank})) \\
& \quad \wedge \texttt{distinct}(\texttt{Y},\texttt{1})
\end{aligned}
$$

Compared to the formula in Figure 6.3, this formula contains a literal of the form $\texttt{true}(f)$ in every disjunct. Thus, we now have the literal $\texttt{true}(\texttt{cell}(\texttt{c},\texttt{1},\texttt{blank}))$ in the first disjunct of ϕ and we add the edge $(\texttt{cell}(\texttt{c},\texttt{1},\texttt{blank}),\texttt{cell}(\texttt{c},\texttt{1},\texttt{x}))$ to the fluent graph instead of $(\emptyset,\texttt{cell}(\texttt{c},\texttt{1},\texttt{x}))$.

6.3.2. Selecting Preconditions for the Fluent Graph

As stated before, several valid fluent graphs exist for a game. For example, both graphs in Figures 6.1 and 6.2 show alternative partial fluents graphs for Tic-Tac-Toe. Arguably, the fluent graph in Figure 6.2 is less informative than the one in Figure 6.1. For example, from Figure 6.1 we can conclude that `cell(c,1,x)` is not reachable from a state in which neither `cell(c,1,x)` itself nor `cell(c,1,blank)` hold. This is valuable information because it essentially says that a cell is lost for the player once it is marked by the opponent. On the other hand, given only Figure 6.2, it seems as if `cell(c,1,x)` can easily be reached within at most two steps from any state of the game because one of the two players is always in control.

We use the fluent graph to compute an admissible distance estimate, i. e., we never overestimate the distance. Thus, we want the distance estimate to be as large as possible. Therefore, we want the paths between nodes in the fluent graph to be as long as possible on average. Selecting the best fluent graph requires to search all possible fluent graphs which is too expensive for almost all games. Additionally, different fluent graphs might be needed to obtain good distance estimates depending on which subgoal of the game, i. e., which fluent, we compute the distance estimate for. Since the construction of fluent graphs is expensive, we construct only one fluent graph for the game. We can influence which fluent graph is constructed by controlling which precondition f of a fluent f' is selected in line 8 of Algorithm 2 for adding the edge (f, f').

If there are several literals of the form $\texttt{true}(f)$ in a disjunct ψ_i of the formula ϕ, we select one of them according to the following heuristics:

1. Only add new edges if necessary. That means, whenever there is a literal $\texttt{true}(f)$ in a disjunct ψ_i such that the edge (f, f') already exists in the fluent graph, we select the fluent f. Thus, no new edge is added to the

fluent graph. The rational of this heuristics is that paths in the fluent graph are longer in average if there are fewer connections between the nodes.

2. Prefer a literal `true`(f) over `true`(g) if f is more similar to f' than g is to f', that is $sim(f, f') > sim(g, f')$. We define the similarity $sim(t, t')$ recursively over ground terms t, t':

$$sim(t,t') = \begin{cases} \sum_i sim(t_i, t'_i) & \text{if } t = f(t_1, \ldots, t_n) \text{ and } t' = f(t'_1, \ldots, t'_n) \\ 0 & \text{if } t \text{ and } t' \text{ differ in function symbol or arity} \\ 1 & \text{otherwise} \end{cases}$$

In human made game descriptions, fluents that are similar typically have strong connections. For example, in Tic-Tac-Toe `cell(c,1,x)` is more related to `cell(c,1,blank)` than to `cell(b,3,x)`. Hence, by using fluents that are similar when adding new edges to the fluent graph, we have a better chance of finding the same fluent again in a different disjunct of ϕ. Thus, we maximize the chance of reusing edges (see item 1).

6.3.3. Overview of the Complete Method

Let us summarize the algorithms and heuristics presented in the previous sections. In Section 6.2.2 (page 72), we defined a distance function based on a fluent graph of the game. We use this distance function in our fuzzy formula evaluation from Chapter 5 to evaluate atoms of the form `true`(f) that occur in the goal and terminal conditions of the game. The distance is defined wrt. to the length of paths between nodes in the fluent graph. To construct a fluent graph from the rules of the game we use Algorithm 2 (page 75). For each fluent f', this algorithm extracts preconditions from a DNF formula ϕ that is an abstraction of `next`(f'). We construct this formula ϕ with Algorithm 4 (page 79). If there are several preconditions for a fluent, we select one according to the heuristics described in Section 6.3.2.

6.4. Examples

Now we discuss two games to show the effectiveness of our approach.

6.4.1. Tic-Tac-Toe

Although, on first sight, Tic-Tac-Toe contains no relevant distance information, we can still take advantage of our distance function. Consider the two states in Tic-Tac-Toe as shown in Figure 6.4. In state s_1, the first row consists of two cells marked with an `x` and a blank cell. In state s_2, the first row contains two `x`s and one cell marked with an `o`. While `xplayer` can immediately win by marking the upper right cell in s_1, this is not possible in s_2. However, the standard evaluation function from Chapter 5 cannot distinguish between these

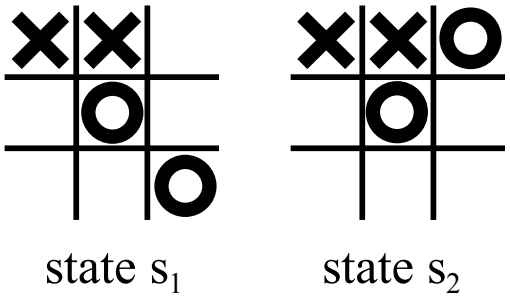

Figure 6.4.: Two states of the Tic-Tac-Toe. While in s_1 the first row is still open, it is blocked in s_2.

two states. Both have two markers in place and one missing for completing the line for the `xplayer`. The condition for `xplayer` completing the first row is:

$$\texttt{true}(cell(a,1,x)) \wedge \texttt{true}(cell(b,1,x)) \wedge true(cell(c,1,x))$$

This condition is evaluated using fuzzy logic as part of the evaluation function defined in Chapter 5. The atoms `true(cell(a,1,x))` and `true(cell(b,1,x))` hold in both states and will be evaluated with $eval(\texttt{true}(\texttt{a},1,\texttt{x})),s) = p$ and $eval(\texttt{true}(\texttt{b},1,\texttt{x})),s) = p$, for both $s = s_1$ and $s = s_2$ and according to Definition 5.2. The atom `true(cell(c,1,x))` holds in neither of the two states. Thus, $eval(\texttt{true}(\texttt{cell}(\texttt{c},1,\texttt{x})),s) = 1 - p$ for both $s = s_1$ and $s = s_2$. However, if all atoms are evaluated with the same value, the overall heuristic value $h(\texttt{xplayer}, s)$ (see Definition 5.3) for `xplayer` will be the same for both states $s = s_1$ and $s = s_2$.

However, we know that a cell marked with `x` or `o` cannot be changed anymore. Hence, while there is still a chance for the `xplayer` to finish the line in s_1, this is not possible in s_2. This fact should be reflected in the heuristics values of s_1 and s_2 and therefore in the evaluation of the atom `true(cell(c,1,x))`.

The distance function defined in this chapter, solves this problem. Based on the fluent graph in Figure 6.1, the distance $\Delta(s_1, \texttt{cell}(\texttt{c},1,\texttt{x}))$ is 1. There is only one step needed from state s_1 to reach a state in which `cell(c,1,x)` holds. However, there is no path from `cell(c,1,o)` nor from any other fluent in s_2 to `cell(c,1,x)` in the fluent graph. Thus, the distance $\Delta(s_2, \texttt{cell}(\texttt{c},1,\texttt{x}))$ is infinite. The new evaluation function that uses the distance estimates defined in Section 6.2.2 evaluates the states s_1 and s_2 differently:

$$\begin{aligned}
eval(\texttt{true}(\texttt{cell}(\texttt{c},1,\texttt{x})),s_1) &= (1-t) * (1 - \delta(s_1, \texttt{cell}(\texttt{c},1,\texttt{x}))) \\
&= (1-t) * \left(1 - \frac{\Delta(s_1, \texttt{cell}(\texttt{c},1,\texttt{x}))}{\Delta_{max}(\texttt{cell}(\texttt{c},1,\texttt{x})) + 1}\right) \\
&= (1-t) * \left(1 - \frac{1}{1+1}\right) = \frac{1-t}{2} > 0 \\
eval(\texttt{true}(\texttt{cell}(\texttt{c},1,\texttt{x})),s_2) &= 0
\end{aligned}$$

Hence, the heuristic value of s_1 is higher than that of s_2: $h(\texttt{xplayer}, s_1) > h(\texttt{xplayer}, s_2)$.

This example shows that the distance estimates are useful even in games where there is no apparent distance information.

6.4.2. Breakthrough

Breakthrough is a two-player game played on a chess board. Like in Chess, the first two ranks contain only white pieces and the last two only black pieces. The pieces of the game are only pawns that move and capture in the same way as pawns in Chess. Whoever reaches the opposite side of the board first wins. Appendix A contains the complete rules for Breakthrough. Figure 6.5 shows the initial position of the game. The arrows indicate the possible moves a pawn can make.

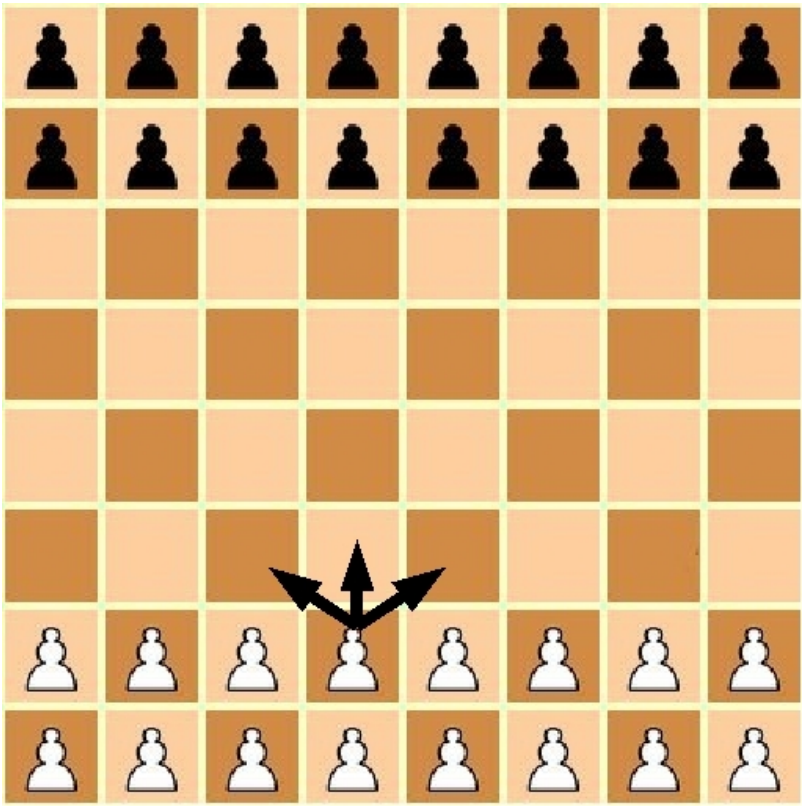

Figure 6.5.: Initial Position in Breakthrough and the move options of a pawn.

The goal condition for the player `black` states that black wins if there is a cell with the coordinates `X,1` and the content `black`, such that `X` is an index ranging from 1 to 8 according to the rules of `index`:

```
1 goal(black, 100) :-
2     index(X), true(cellholds(X, 1, black)).
```

Grounding this rule yields the following 8 instances:

```
1 goal(black,100) :- index(1),true(cellholds(1,1,black)).
```

```
goal(black,100) :- index(2),true(cellholds(2,1,black)).
goal(black,100) :- index(3),true(cellholds(3,1,black)).
goal(black,100) :- index(4),true(cellholds(4,1,black)).
goal(black,100) :- index(5),true(cellholds(5,1,black)).
goal(black,100) :- index(6),true(cellholds(6,1,black)).
goal(black,100) :- index(7),true(cellholds(7,1,black)).
goal(black,100) :- index(8),true(cellholds(8,1,black)).
```

The plain heuristics from Chapter 5 without any distance evaluation does not differentiate any of the states in which the goal is not reached because `true(cellholds(X, 1, black))` is false in all of these states for any instance of `X`. However, a state where the black pawns are more advanced is typically better.

Figure 6.6 shows a part of the fluent graph that our algorithm generates for Breakthrough. As one can clearly see, the fluent graph and, thus, the distance function captures the way pawns move. Thus, states where black pawns are nearer to one of the cells `(1,8)`, ..., `(8,8)` are preferred. Hence, a state evaluation function with the distance function proposed in this chapter is effective.

Moreover, the fluent graph and, thus, the distance function, contains the information that some locations are only reachable from certain other locations. For example, cell `(1,1)` is not reachable from cell `(3,2)` because sideways or backward moves are not possible. Together with the evaluation function this leads to what could be called "strategic positioning": states with pawns on the side of the board are worth less than those with pawns in the center. This is due to the fact that a pawn in the center may reach more of the 8 possible destinations than a pawn on the side. Thus, if a pawn is in the center of the board, more of the atoms `true(cellholds(X, 1, black))` will be evaluated with a higher value, because the distances to the respective cells are lower. Since the atoms `true(cellholds(X, 1, black))` occur only positive in the goal condition, a higher evaluation of each of them will lead to a higher heuristic value $h(\texttt{black}, s)$ of the state.

6.5. Evaluation

For evaluation, we prototypically implemented our distance function and equipped Fluxplayer with it. We then set up this version of Fluxplayer ("flux_distance") against its version without the new distance function ("flux_basic"). We used the version of Fluxplayer that came in 4th in the 2010 championship. This version of Fluxplayer uses the evaluation function described in Chapter 5, but also includes many optimizations and hand-crafted heuristics, some of which are described later in this thesis. Since flux_basic is already endowed with a distance heuristic, the evaluation is comparable to a competition setting of two competing heuristics using distance features.

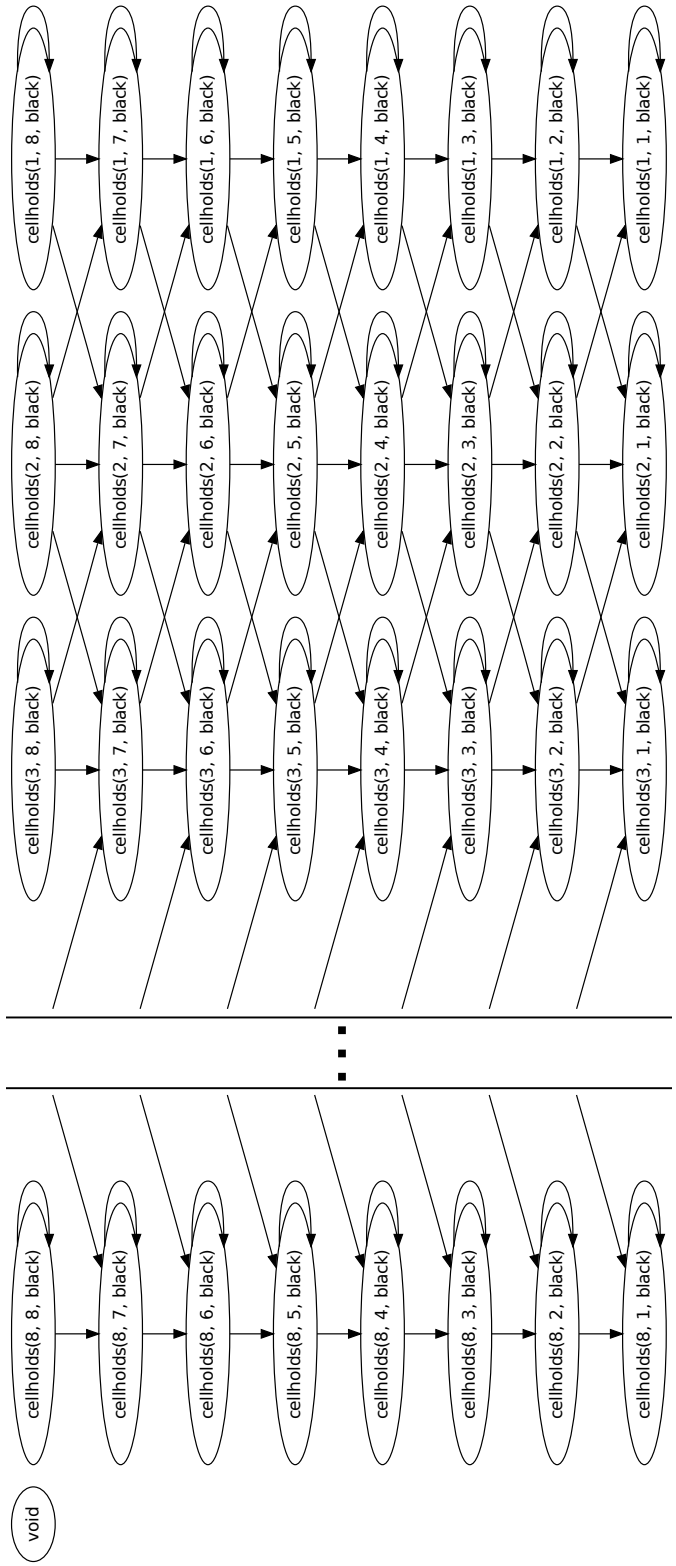

Figure 6.6.: A partial fluent graph for Breakthrough.

We chose 19 games for comparison for which we conducted 100 matches on average per game. Figure 6.7 shows the results.

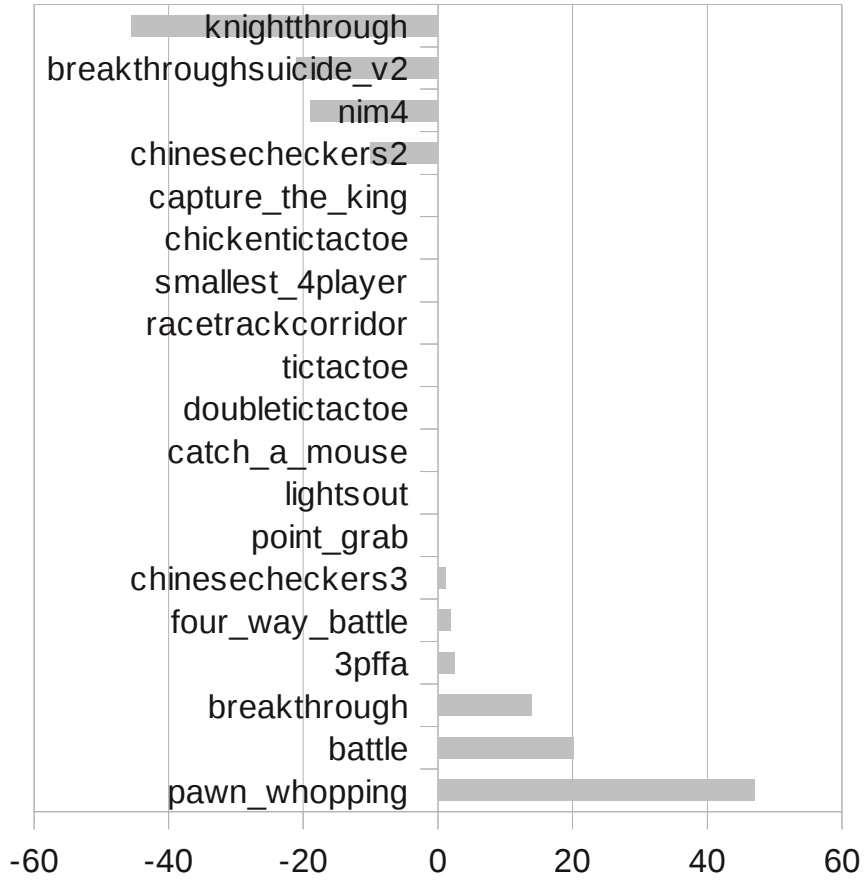

Figure 6.7.: Difference of the win rates of flux_distance (Fluxplayer with the new distance function) and flux_basic (the 2010 version of Fluxplayer). Positive numbers indicate an advantage of flux_distance.

The values indicate the difference in win rate, e. g., a value of +10 means that flux_distance won 55% of the games against flux_basic winning 45%. Thus, positive values indicate an advantage of flux_distance while negative values indicate an advantage of flux_basic.

Obviously, the proposed heuristics produces results comparable to the flux_basic heuristics, with both having advantages in some games. This has several reasons, most importantly however the distance function presented in this chapter, at least in the way it is implemented now, is more expensive than the distance estimation used in flux_basic. Therefore, the evaluation of a state takes longer and the search tree cannot be explored as deeply as with cheaper heuristics. This accounts for three of the four underperforming games:

- In nim4, the flux_basic distance estimation provides essentially the same results as our new approach. However, our new approach is more expensive

and, thus, only a smaller portion of the game tree could be searched.
- In chinesecheckers2 and knightthrough, the new distance function is actually more accurate than the old one. However, it slows down the search more than its better accuracy can compensate.

The fourth game in which flux_distance looses against flux_basic is breakthrough_suicide. The game is exactly the same as breakthrough with swapped goals, that is, the player who reaches the opposite side of the board first loses. The distance estimates in breakthrough are more accurate with the new distance function, as can be seen from the fact that flux_distance wins against flux_basic in breakthrough. However, the evaluation function of Fluxplayer is not well suited for this game. The more accurate distance estimates seem to amplify the bad evaluation function. Therefore, flux_distance plays the game even worse than flux_basic.

We can see that the new distance estimate improves the performance in breakthrough, battle and pawn_whopping. For the remaining games, we did not observe a significant difference between the two systems.

Finally, in some of the games, no changes were found because both distance estimates performed equally well. However, in some of these games rather specific heuristics and analyzation methods of flux_basic could be replaced by the more general approach presented in this chapter. For example, flux_basic contains a special method to detect when a fluent is unreachable, while this information is automatically included in our distance estimate, as shown in Section 6.4.1. Also, the distance function described in this chapter can replace the less general method of detecting boards and quantities that we presented in Chapter 5.

6.6. Future Work

The main problem of the approach is its computational cost for constructing the fluent graph. The most expensive steps of the fluent graph construction are the grounding of the DNF formulas ϕ and processing the resulting large formulas to select edges for the fluent graph. For many complex games, these steps cause either out-of-memory errors or take longer than the start clock of the match. Thus, an important line of future work is to reduce the size of formulas before the grounding step and without losing relevant information.

One way to reduce the size of ϕ is a more selective expansion of predicates (line 5) in Algorithm 4. Developing heuristics for this selection of predicates is one of the goals for future research.

Alternatively, we suggest to construct fluent graphs from non-ground representations of the preconditions of a fluent to skip the grounding step completely. For example, the partial fluent graph in Figure 6.1 is identical to the fluent graphs for the other 8 cells of the Tic-Tac-Toe board. The fluent graphs for all 9 cells are obtained from the same rules for `next(cell(X,Y,_)`, just with different

instances of the variables `X` and `Y`. By not instantiating `X` and `Y`, the generated DNF is exponentially smaller while still containing the same information.

The quality of the distance estimates depends mainly on which preconditions are selected as edges for the fluent graph (see Section 6.3.2). At the moment the heuristics we use for this selection are rather ad hoc. A further point for future work is to investigate how these heuristics can be improved.

6.7. Summary

We have presented an approach to compute an admissible estimate of the distance $\Delta(s, f)$ between a state s and a fluent f, that is, an estimate of the minimal number of steps needed to fulfil a fluent f starting in a state s. The approach is based on extracting causal dependencies between fluents from the game rules. Thus, the complexity of the algorithm does not depend on the size of the game tree, but only on the size of the game description. Game descriptions are typically exponentially smaller than the game tree. We showed how the new distance estimate can be used in an evaluation function to evaluate atomic subgoals of the game. We also showed that the method is indeed effective for different types of games.

The approach presented in this chapter is more accurate than any approach presented before because patterns in which fluents change (e. g., the moving pattern of a piece) are considered: In contrast to standard Manhattan distance evaluation, our new approach produces different distance functions depending on whether the underlying piece is, e. g., a knight, a bishop or a pawn. Furthermore, we are able to detect infinite distances which correspond to non-reachability of fluents. An adequate use of this information can improve the state evaluation function.

The drawback of the more accurate distance function is that it is defined on the ground instances of the GDL formulas obtained from the game rules (last step of Algorithm 4). Although conceptually simple, for game rules with many variables, this step may use a lot of time and memory. The current prototypical implementation takes between fractions of a second for simple games such as Tic-Tac-Toe, up to several minutes and gigabytes of memory for complex games, such as Checkers. We presented ideas for future work addressing this issue.

7. Proving Properties of Games

Knowledge-based GGP systems, such as Kuhlplayer [KDS06], Cluneplayer [Clu07], or Fluxplayer, rely on the ability to automatically extract game-specific knowledge from the rules of a game. This knowledge serves a variety of purposes that are crucial for good game play:

- Games need to be classified in order to choose the right search method. For example, Minimax with alpha-beta pruning is only suitable for two-player, zero-sum, and turn-taking games.
- Recognition of structures like boards, pieces, and mobility of pieces is needed to automatically construct good evaluation functions for the assessment of intermediate states.
- Game-specific knowledge can be used to cut off the search in states that are provably lost for the player.

While existing systems extract this kind of knowledge, they do not actually attempt to prove it; rather they generate random sample matches to test whether a property is violated at some point, and then rely on the correctness of this informed guess. The first method for automatically proving properties for general games is presented in [RvdHW09]. But this method requires to systematically search the entire set of reachable states in a game and therefore is not suitable for practical play. Finding a practical method of rigorously proving game-specific knowledge from the mere rules of a game is an open and challenging problem in General Game Playing.

In this chapter, we present a solution to this problem in the form of a method which allows systems to automatically prove properties that hold across all reachable states. We show how the focus on this kind of properties allows us to reduce the automated theorem proving task to a simple proof of an induction step and its base case.

We will use the paradigm of Answer Set Programming (ASP) introduced in Section 2.5 to validate properties of games that are specified in the general Game Description Language (GDL). The advantage of using ASP is the availability of efficient off-the-shelf systems that we can use for the automatic proofs.

In the next section, we will show how the concept of answer sets can be applied in the context of GDL in order to systematically prove game-specific properties. Then, the correctness of this method will be formally proved. Furthermore, we will report on experiments with an off-the-shelf ASP system [oP10] in combination with our general game player, Fluxplayer. The results from this chapter were published in [ST09a].

7.1. Proving Game Properties Using Answer Set Programming

We address the following challenge in this chapter: given a GDL description of an unknown game, how can a general game player prove fully automatically game-specific knowledge in form of properties that hold across all finitely reachable states?

As an example, recall the formal description of Tic-Tac-Toe given in Figure 2.1. These rules and their semantics according to Definition 2.14 imply that the argument of the feature $control(P)$ is unique in every reachable state. Intuitively, this means that only one player is in control in every state. The ability to derive this fact is essential for a general game player to be able to identify Tic-Tac-Toe as a turn-taking game. A similar but less obvious consequence of the given description is the uniqueness of the third argument of $cell(X, Y, C)$ in every reachable state. That is, the fact that the first two arguments of *cell* are input arguments (see Definition 5.4). This knowledge may help a general game player to identify this feature as representing a two-dimensional "board" with "markers" C. We already find such properties, as described in Section 5.2. However, the method described there does not guarantee correct results.

As long as a game is finite, properties of this kind can in principle be determined by a complete search through the state transition diagram for a game [RvdHW09]. However, for games that are simple enough to make this practically feasible, a general game player does not actually need game-specific knowledge because it can solve the game by exhaustive search anyway. For this reason, the challenge for the practice of General Game Playing (GGP) is to develop a local proof method for properties. In case of game-specific properties that hold across all reachable states, the key idea is to reduce the automated theorem proving task to a simple proof of an induction step and its base case.

In the specific setting of GGP, proving a property φ by induction means to show that

1. φ holds in the initial state, and
2. if φ holds in a state and all players choose legal moves, then φ holds in the next state, too.

Because game descriptions in GDL are logic programs with negation, this general proof principle can be put into practice with the help of the Answer Set Programming (ASP) paradigm. Answer sets are models of logic programs with negation according to Definition 2.20.

As an example, consider the following simple program that can be used to proof that base case of the property that there is exactly one player in control in every state of Tic-Tac-Toe.

```
1 init(cell(a,1,blank)). ... init(cell(c,3,blank)).
2 init(control(xplayer)).
3
4 cdom(xplayer).
```

```
5 cdom(oplayer).
6 phi_init :- 1 { init(control(X)) : cdom(X) } 1.
7 :- phi_init.
```

The program contains the rules describing the initial state of Tic-Tac-Toe. In addition, there are two rules defining the predicate `cdom`, which encodes the domain of the `control` fluent. The rule in line 6 defines the predicate `phi_init` with the help of a weight atom. The rule states that `phi_init` holds if there is exactly one instance of `init(control(X))` that satisfies `cdom(X)`, that is, there is exactly one player who has control in the initial state, either `xplayer` or `oplayer`. Finally, the constraint in line 7 ensures that `phi_init` does not hold in any answer set. Since `phi_init` is clearly entailed by the definition of the initial state, that means that the above program has no answer set. This is a *proof* of the fact that exactly one instance of $control(X)$ holds in the initial state according to the rules of Tic-Tac-Toe.

In a similar fashion, we can prove the induction step for this uniqueness property of fluent `control(P)` by adding the rules from Figure 7.1 to the game description of Tic-Tac-Toe depicted in Figure 2.1. In Figure 7.1, lines 1–11 provide an appropriate definition of the domains for the fluents and the moves, respectively, as determined by the algorithm presented in Section 5.2.1. The domain of the fluents is the union of the domains of `init` and `next`, that is, all ground terms that may occur as an argument of `init` or `next`. The domain of the moves is the domain of the second argument of `legal`. Line 13 allows for arbitrary instances of the given fluents to hold in a current state. Line 15 requires every player to select exactly one move, while line 16 excludes any move that is not legal.

The induction hypothesis is axiomatised in lines 18 and 19: There is exactly one instance of $control(X)$ in the current state. Finally, lines 21 and 22 together encode the negation of the "theorem" that this uniqueness holds in the next state, too. Again, the program admits no answer set, which proves the claim that there is exactly one instance of $control(X)$ in every reachable state of Tic-Tac-Toe.

An interesting aspect of inductively proving properties of general games can be observed when trying to verify the uniqueness of the third argument of $cell(X, Y, C)$ in the same way. The straightforward attempt produces a counter-example to the induction step, namely, an answer set containing

```
true(cell(a,1,blank)),
true(control(xplayer)),
true(control(oplayer)),
does(xplayer,mark(a,1)),
does(oplayer,mark(a,1)),
next(cell(a,1,x)), next(cell(a,1,o))
```

In this model, cell $(a, 1)$ has a unique content in the current state but then gets marked by both players simultaneously, which is a perfectly legal joint move under the assumption that each of the two players has the control. This

```
dom_control(xplayer). dom_control(oplayer).
dom_cell1(a). dom_cell1(b). dom_cell1(c).
dom_cell2(1). dom_cell2(2). dom_cell2(3).
dom_cell3(x). dom_cell3(o). dom_cell3(blank).

dom_fluent(control(X)) :- dom_control(X).
dom_fluent(cell(X,Y,C)) :-
    dom_cell1(X), dom_cell2(Y), dom_cell3(C).

dom_move(mark(X,Y)) :- dom_cell1(X), dom_cell2(Y).
dom_move(noop).

0 { true(F) : dom_fluent(F) }.

1 { does(R,M) : dom_move(M) } 1 :- role(R).
:- does(R,M), not legal(R,M).

phi_true :- 1 { true(control(X)) : dom_control(X) } 1.
:- not phi_true.

phi_next :- 1 { next(control(X)) : dom_control(X) } 1.
:- phi_next.
```

Figure 7.1.: Answer set program for the induction step of the proof that exactly one instance of $control(X)$ holds in every reachable state.

shows that a proof may require to incorporate previously derived knowledge. The above answer set—and other, similar ones—disappear when one adds the assumption that in the current state exactly one instance of $control(X)$ holds. In the following section, we describe this proof method in general and show its correctness under the semantics of GDL as given in Definition 2.14.

7.2. The General Proof Method and its Correctness

Our proof method automatically proves that all finitely reachable states in a game satisfy a property φ. Before we introduce the general proof method, let us formally define what a state property is and what it means for a property to be satisfied by a state.

Definition 7.1 (State Property)**.** *Let D be a valid GDL specification whose signature determines the set of ground terms Σ. A* state property *is a first-order formula φ over a signature whose ground atoms are from Σ. In addition to the usual logical connectives and quantifiers, we allow the counting quantifier $(\exists_{l..u}\vec{x})\phi(\vec{x})$, with the intended meaning that at least l and at most u different ground instances of $\phi(\vec{x})$ hold.*

Definition 7.2 (State Property Satisfied by a State)**.** *Let D be a valid GDL specification whose signature determines the set of ground terms Σ, Γ be the game for D, and $\mathcal{S}$ be the set of states of Γ (cf. Definition 2.14). Let $s \in \mathcal{S}$ be a state in of Γ. The notion of s* satisfying *the state property φ (written: $s \models \varphi$) is defined inductively as follows:*

$$\begin{array}{rll}
s \models f & \textit{iff} & f \in s \textit{ (where } f \textit{ atomic and ground)} \\
s \models \varphi_1 \wedge \varphi_2 & \textit{iff} & s \models \varphi_1 \textit{ and } s \models \varphi_2 \\
s \models \varphi_1 \vee \varphi_2 & \textit{iff} & s \models \varphi_1 \textit{ or } s \models \varphi_2 \\
s \models \neg\varphi & \textit{iff} & s \not\models \varphi \\
s \models (\exists x)\varphi(x) & \textit{iff} & s \models \varphi(t) \textit{ for some ground term } t \in \Sigma \\
s \models (\exists_{l..u}\vec{x})\varphi(\vec{x}) & \textit{iff} & s \models \varphi(\vec{t}) \textit{ for at least } l \textit{ and at most } u \textit{ different } \vec{t} \in \Sigma^n, \\
 & & \textit{where } \vec{x} = (x_1, \ldots, x_n)
\end{array}$$

The remaining logical connectives are defined as the usual macros. For example,

$$\begin{array}{rcl}
\varphi_1 \supset \varphi_2 & \stackrel{\text{def}}{=} & \neg\varphi_1 \vee \varphi_2 \\
(\forall x)\varphi(x) & \stackrel{\text{def}}{=} & \neg(\exists x)\neg\varphi(x)
\end{array}$$

When employing Answer Set Programming to automatically prove that all finitely reachable states in a game satisfy a property φ, a general game player proceeds as follows.

Let D be a given GDL specification. The use of ASP requires additional clauses *Dom* without negation that define the domains of the features and moves according to D using predicates `dom_fluent` and `dom_move`, respectively. A suitable *Dom* can be easily computed on the basis of the domain graph for D (see Section 5.2.1).

Furthermore, for every predicate $p \in \{\texttt{init}, \texttt{true}, \texttt{next}\}$ let φ^p be an atom that, together with an associated finite set of stratified clauses Φ^p, encodes the fact that the property φ is satisfied in the state represented by keyword p. For example, $\varphi^{\texttt{init}}$ encodes the fact the φ is satisfied by the initial state of the game, that is, by the state $\{f : D \models \texttt{init}(f)\}$.

Note that in games with finitely many fluents such an encoding always exists: any φ can in principle be represented by an exhaustive propositional formula, although in practice a compact encoding (as in the examples in the preceding section) is desirable.

The automatic proof that φ holds in all reachable states in the game described by D is then obtained in two steps:

Base Case Show that there is no answer set for $D \cup Dom$ augmented by

$$\begin{array}{l}
\Phi^{\texttt{init}} \\
\texttt{:-}\ \varphi^{\texttt{init}}.
\end{array} \tag{7.1}$$

Induction Step Suppose that ψ is a (possibly empty) conjunction of state properties that have been proved earlier to hold across all reachable game states. Show that there is no answer set for $D \cup Dom$ augmented by

```
0 { true(F) : dom_fluent(F) }.
1 { does(R,M) : dom_move(M) } 1 :- role(R).
:- does(R,M), not legal(R,M).
Ψ^true
:- not ψ^true.                                   (7.2)
Φ^true
:- not φ^true.
Φ^next
:- φ^next.
```

The correctness of this general proof method can be shown with the help of the following three theorems.

The first theorem states that if a state property φ is not satisfied in a reachable state, the property is either not satisfied in the initial state of the game or there is a state transition such that φ holds in one state but not in the successor state. Effectively, this theorem states that proving properties by induction is indeed correct.

Theorem 7.1. *Let $\Gamma = (R, s_0, T, l, u, g)$ be a game with states $\mathcal{S}$ and actions $\mathcal{A}$ as defined in Definition 2.1. Let φ be a state property. If there is a finitely reachable state in $\mathcal{S}$ which does not satisfy φ, then*

1. *$s_0 \not\models \varphi$, or*
2. *there is a finitely reachable state $s \in \mathcal{S}$ and a joint action $A : R \to \mathcal{A}$ such that*
   - *$s \models \varphi$;*
   - *$l(r, A(r), s)$ for all $r \in R$; and*
   - *$u(A, s) \not\models \varphi$.*

Proof. Reachability means that successively, starting in state s_0, a joint action for the roles is chosen that is legal according to l, and then the current state is updated according to u (cf. Definition 2.4). Given that a finitely reachable state s' exists that violates φ, there is a sequence of states starting in s_0 that leads to s'. This sequence of states must contain a first state that violates φ. This state is either s_0 or has a predecessor that satisfies φ, which implies the claim. □

Next, we state the correctness of the base case and the induction step of the induction proof.

Theorem 7.2 (Correctness of Base Case)**.** *Consider a valid GDL specification D whose semantics is (R, s_0, T, l, u, g). Let Dom be a program without negation defining the domains for the fluents and moves according to D. For any state property φ for which $D \cup Dom \cup \{(7.1)\}$ does not admit an answer set, we have that $s_0 \models \varphi$.*

Proof. We prove that if $s_0 \not\models \varphi$ then $D \cup Dom \cup \{(7.1)\}$ admits an answer set. Since $D \cup Dom$ is stratified, it admits a unique answer set M that coincides with its standard model [GL88]. Hence, $s_0 \not\models \varphi$ implies $M \not\models \varphi^{\texttt{init}}$. This in turn implies that M is also an answer set for $D \cup Dom \cup \{(7.1)\}$ (in which $\varphi^{\texttt{init}}$ is false). □

Theorem 7.3 (Correctness of Induction Step)**.** *Consider a valid GDL specification D whose semantics is (R, s_0, T, l, u, g). Let Dom be a positive program defining the domains for the fluents and moves according to D. For any state properties Φ and φ for which $D \cup Dom \cup \{(7.2)\}$ does not admit an answer set, there does not exist a state $s \in \mathcal{S}$ and a joint action $A : R \to \mathcal{A}$ such that*

1. *$s \models \psi$ and $s \models \varphi$;*
2. *$(r, A(r), s) \in l$ for all $r \in R$; and*
3. *$u(A, s) \not\models \varphi$.*

Proof. We prove that $D \cup Dom \cup \{(7.2)\}$ admits an answer set whenever there exists a state s and a joint action A that satisfy the given conditions 1–3. Since $D \cup Dom \cup s^{\texttt{true}} \cup A^{\texttt{does}}$ is stratified, it admits a unique answer set M. According to conditions 1 and 2, M is also an answer set for $D \cup Dom$ augmented by the following clauses from (7.2).

```
0 { true(F) : dom_fluent(F) }.
1 {does(R,M) : dom_move(M) } 1 :- role(R).
:- does(R,M), not legal(R,M).
Ψ^true
:- not ψ^true.
Φ^true
:- not φ^true.
```

Because condition 3 implies that $M \not\models \varphi^{\texttt{next}}$, model M is also an answer set for $D \cup Dom \cup \{(7.2)\}$ (in which $\varphi^{\texttt{next}}$ is false). □

Finally, we state the correctness of the proof procedure.

Theorem 7.4 (Correctness of the Proof Procedure)**.** *Consider a valid GDL specification D whose semantics is (R, s_0, T, l, u, g). Let Dom be a positive program defining the domains for the fluents and moves according to D. Let φ be a state property and ψ be a conjunction of state properties that are known to hold in all reachable states of the game. If neither $D \cup Dom \cup \{(7.1)\}$ nor $D \cup Dom \cup \{(7.2)\}$ admit an answer set then the property φ holds across all reachable states.*

Proof. If condition 1 in Theorem 7.1 is satisfied, then $D \cup Dom \cup \{(7.1)\}$ must admit an answer set according to Theorem 7.2. If condition 2 in Theorem 7.1 is satisfied, then $D \cup Dom \cup \{(7.2)\}$ must admit an answer set under the assumption that all reachable states satisfy ψ (Theorem 7.3). Hence, if neither is the case then the property φ must hold across all reachable states. □

7.3. An Automated Theorem Prover

The above method is implemented in our GGP system Fluxplayer using Clingo [oP10] as answer set solver. The answer set programs for base case and induction step are automatically generated from the game description.

We currently try to prove the following properties:

Input/output arguments We consider sets of input arguments as defined in Definition 5.4. For every fluent f and every set of argument indices I of f we try to prove that I is a set of input arguments of f. Two examples in the game of Tic-Tac-Toe were presented in Section 7.1: Proving that $I = \emptyset$ is a set of input arguments of *control* and proving that $I = \{1, 2\}$ is a set of input arguments of *cell*.

These properties can be used for the heuristics as described in Chapter 5. They are also important as additional information (properties ψ) when proving other properties.

In general, if $I = \{i_1, \ldots, i_m\} \subseteq \{1, \ldots, n\}$ and $O = \{i_{m+1}, \ldots, i_n\} = \{1, \ldots, n\} \setminus I$ are sets of argument indices of an n-ary fluent f, the property ϕ stating that I is a set of input arguments of f (an O is a set of output arguments) is described by the following formula:

$$\begin{aligned} \phi &= (\forall x_{i_1}, \ldots, x_{i_m})(\exists_{0..1} x_{i_{m+1}}, \ldots, x_{i_n}) f(x_1, \ldots, x_n) \\ &= \neg(\exists x_{i_1}, \ldots, x_{i_m}) \neg(\exists_{0..1} x_{i_{m+1}}, \ldots, x_{i_n}) f(x_1, \ldots, x_n) \end{aligned}$$

Without loss of generality, let $I = \{1, \ldots, m\}$ and $O = \{m+1, \ldots, n\}$. Then this property ϕ can be encoded by the following set of clauses $\Phi^{\texttt{true}}$.

```
1 phi_true(X1, ..., Xm) :-
2     0 { true(f(X1, ..., Xn))
3         : dom_m+1(Xm+1) : ...
4         : dom_n(Xn) } 1.
5 not_phi_true :-
6     dom_1(X1), ..., dom_m(Xm),
7     not phi_true(X1, ..., Xm).
8 phi_true :- not not_phi_true.
```

Where the predicates `dom_i` refer to the predicates in *Dom* encoding the domain of the i-th argument of f.

Zerosum game We try to prove that a game is a zerosum game, that is, that the goal values of all players add up to the same sum in all terminal states. With a slight abuse of notation, the property $\phi_{zerosum}$ can be defined as follows:

$$\begin{aligned} \phi_{zerosum} \stackrel{\text{def}}{=} & (\forall v_1, \ldots, v_n)\, \texttt{terminal} \wedge \texttt{goal}(r_1, v_1) \wedge \ldots \wedge \texttt{goal}(r_n, v_n) \\ & \supset\ v_1 + \ldots + v_n = sum \end{aligned}$$

Here, $r_1, \ldots, r_n$ are the roles of the game and *sum* is a number that is computed by generating one reachable terminal state of the game and adding the goal values of all roles in this state. A reachable terminal state can be easily found by executing an arbitrary sequence of legal joint moves until a terminal state is reached.

In $\phi_{zerosum}$, the terms `terminal` and `goal`(r_i, v_i) stand for suitable state properties derived from the terminal and goal rules, respectively. This can be achieved by expanding all predicates in the bodies of those rules and replacing every occurrence of an atom `true`(f) with f. Furthermore, the term $v_1 + \ldots + v_n = sum$ stands for a suitable encoding of the fact that $v_1, \ldots, v_n$ add up to *sum* as a first-order formula. Such an encoding exists because there are only finitely many possible goal values in a game.

Our proof method, which is depicted in Algorithm 5, conducts a systematic search for all state properties of the above forms that hold in a game.

Algorithm 5 Algorithm for the systematic search for all state properties that hold in a game.

Input: Φ_{all}, the set of properties we want to prove
Output: Ψ_{proved}, the set of properties that are proved

1: $\Phi_{proved} := \emptyset$
2: Set the set of proofs to attempt $\Phi_? := \Phi_{all}$
3: **while** $\Phi_? \neq \emptyset$ **do**
4: Select a property $\varphi \in \Phi_?$
5: $\Phi_? := \Phi_? \setminus \{\varphi\}$
6: Try to prove φ using $\psi = \bigwedge_{\varphi \in \Phi_{proved}} \varphi$
7: **if** φ could be proved **then**
8: $\Phi_{proved} := \Phi_{proved} \cup \{\varphi\}$
9: $\Phi_? := \Phi_{all} \setminus \Phi_{proved}$
10: **end if**
11: **end while**

Given a set of state properties Φ_{all} that we are interested in, we try to prove every property $\phi \in \Phi_{all}$ in turn and use the conjunction of the properties Ψ_{proved} that were already proved as the formula ψ in the proof procedure presented in Section 7.2. Whenever some property is proved we add it to Φ_{proved} and restart the procedure (cf. line 9) with the properties that could not be proved yet.

7.4. Optimizations

The proof method described in the previous section can be improved in the following ways:

- The time and memory requirements for each single proof can be reduced by reducing the size of the generated answer set programs.
- The number of restarts can be reduced by selecting the properties φ in line 4 in a good order.

- Reducing the number of properties to prove, i. e., the set Φ_{all}, before the proof attempts by removing those properties for which we can deduce that they do not hold.

We will present ways how to address all these points in the following subsections.

7.4.1. Reducing the Size of Generated Answer Set Programs

The sizes of the answer set programs that are used in the base case ($D \cup Dom \cup \{(7.1)\}$) and induction step ($D \cup Dom \cup \{(7.2)\}$) for proving a property φ have a great influence on time and memory requirements of the answer set solver. The reason is that, today's answer set solvers work on essentially propositional programs, that is, answer set programs (ASPs) have to be grounded before computing the answer sets. To ground an ASP, every rule in the program is replaced by every ground instance of this rule. The number of ground instances of a rule is exponential in the number of variables in the rule. Thus, the size of the grounded ASP is exponential in the size of the original non-grounded ASP, in the worst case. This causes high time and memory requirements for both grounding and solving ASPs for games with non-trivial rules.

For the base case and induction step proofs, we are only interested in the existence (or non-existence) of an answer set for the generated ASP. Thus, we can replace an ASP P with a smaller ASP P' such that P' has an answer set iff P does. In our implementation we compute P' from P by removing all rules from P that do not influence the existence of an answer set.

As an example, consider the rules for the induction step of the proof that there is exactly one fluent *control*(X) in every state of Tic-Tac-Toe. The ASP $P = D \cup Dom \cup \{(7.2)\}$ consists of the game description D of Tic-Tac-Toe (Figure 2.1) and the rules $Dom \cup \{(7.2)\}$ in Figure 7.1. Now, it is easy to see that the rules for `goal` and `terminal` are irrelevant for the existence of an answer set of P because the predicates `goal` and `terminal` do not occur anywhere else in the program P. Likewise, the rules for `next(cell(X,Y,C))` (lines 15–20) are irrelevant because there is no instance of `next(cell(X,Y,C))` anywhere else in P.

This approach is theoretically based on the splitting set theorem [LT94]. Informally, the splitting set theorem states that an answer set program P can be split into two programs $b_U(P)$ and $P \setminus b_U(P)$ called bottom and top of P, respectively. Every union of answer sets of the bottom and the top program is an answer set of P. Let us repeat the formal definitions from [LT94]:

Definition 7.3 (Splitting Set)**.** *A splitting set U for a program P is any set U of literals such that, for every rule $r \in P$, if $head(r) \cap U \neq \emptyset$ then $lit(r) \subseteq U$, where $head(r)$ denotes the set of literals in the head of r and $lit(r)$ denotes the set of all literals in r. The set of rules $b_U(P) = \{r \in P | lit(r) \subseteq U\}$ is called bottom of P and $P \setminus b_U(P)$ is called top of P with respect to U.*

Definition 7.4 (Solution to P)**.** *Let U be a splitting set of P and X be a set of literals. Let $pos(r)$ denote the set of positive literals in the body of rule r, $neg(r)$*

denote the set of negative literals in the body of r, and $body(r) = pos(r) \cup neg(r)$. The program $e_U(P, X)$ consists of all rules r' with $head(r') = head(r)$ and $body(r') = body(r) \setminus U$ for some $r \in P$ such that $pos(r) \cup U \subseteq X$ and $neg(r) \cup U = \emptyset$.

A solution to P with respect to a splitting set U is a pair $< X, Y >$ of sets of literals such that

- *X is an answer set for $b_U(P)$,*
- *Y is an answer set for $e_U(P \setminus b_U(P), X)$, and*
- *$X \cup Y$ is consistent, i. e., does not contain the literals p and $\neg p$ for any atom p.*

Theorem 7.5 (Splitting Set Theorem)**.** *Let U be a splitting set for a logic program P. A set A of literals is a consistent answer set for P if and only if $A = X \cup Y$ for some solution $< X, Y >$ to P with respect to U.*

For proving properties using answer set programming, we compute the splitting set U as the smallest splitting set of P that contains all literals from $Dom \cup \{(7.1)\}$ or $Dom \cup \{(7.2)\}$, respectively. Thus the bottom $b_U(P)$ of the answer set program contains all rules in $Dom \cup \{(7.1)\}$ (or $Dom \cup \{(7.2)\}$) and all rules from the game description D whose heads occur in a rule $Dom \cup \{(7.1)\}$ (or $Dom \cup \{(7.2)\}$).

In our Tic-Tac-Toe example, U contains all instances of

- `role(R)`,
- `legal(R,M)`,
- `next(control(X))`,
- `true(F)`,
- `does(R,M)`, and
- all domain predicate `dom_control(C)`, ..., `dom_move(M)`.

Thus, $b_U(P)$ is the following program:

```
role(xplayer). role(oplayer).

legal(P, mark(X,Y)) :-
    true(control(P)), true(cell(X,Y,blank)).
legal(P,noop) :-
    role(P), not true(control(P)).

next(control(oplayer)) :- true(control(xplayer)).
next(control(xplayer)) :- true(control(oplayer)).

dom_control(xplayer). dom_control(oplayer).
dom_cell1(a). dom_cell1(b). dom_cell1(c).
dom_cell2(1). dom_cell2(2). dom_cell2(3).
dom_cell3(x). dom_cell3(o). dom_cell3(blank).
dom_fluent(control(X)) :- dom_control(X).
dom_fluent(cell(X,Y,C)) :-
    dom_cell1(X), dom_cell2(Y), dom_cell3(C).
```

```
18 dom_move(mark(X,Y)) :- dom_cell1(X), dom_cell2(Y).
19 dom_move(noop).
20
21 0 { true(F) : dom_fluent(F) }.
22
23 1 { does(R,M) : dom_move(M) } 1 :- role(R).
24 :- does(R,M), not legal(R,M).
25
26 phi_true :- 1 { true(control(X)) : dom_control(X) } 1.
27 :- not phi_true.
28
29 phi_next :- 1 { next(control(X)) : dom_control(X) } 1.
30 :- phi_next.
```

We claim that to decide whether P has an answer set, it is sufficient to decide whether $b_U(P)$ has an answer set.

Theorem 7.6 (Correctness of the Reduction)**.** *Let P be an answer set program of the form $D \cup Dom \cup Q$ with $Q = \{(7.1)\}$ or $Q = \{(7.2)\}$. Let U be the smallest splitting set such that U contains all literals from Q.*

The program P admits an answer set A if and only if $b_U(P)$ admits to an answer set X and $X \subseteq A$.

Proof. The top of P, i. e., $P \backslash b_U(P)$, contains only rules from the game description D and is thus stratified. Hence, for any set X, $e_U(P \setminus b_U(P), X)$ is stratified and admits an answer Y. If X is an answer set for $b_U(P)$ it does not contain any literals from the head of $P \setminus b_U(P)$. Thus X and Y are consistent. Hence, the conditions 2 and 3 of Definition 7.4 are fulfilled if X is an answer set for $b_U(P)$. Thus, according to the splitting set theorem, $A = X \cup Y$ is an answer set for P if and only if X is an answer set for $b_U(P)$ and Y is the answer set for $e_U(P \setminus b_U(P), X)$. □

As the result of this theorem, for our proof method it is sufficient to check whether $b_U(P)$ has an answer set. Since $b_U(P)$ is typically much smaller than P, this results in a reduction of time and memory requirements of our proof procedure.

7.4.2. Improved Domain Calculation

For formulating state properties as well as for the encoding of the action and state generators that are used in the proofs, we need information about the domains of predicates and functions in a game description. Specifically, we need the set of potential actions *ADom* of a game to encode the domain of move `dom_move` and the set of all ground fluents *FDom* to encode the domain of fluent `dom_fluent` for the induction step of the proof (cf. (7.2)).

In principle the exact domains can be computed: The minimal set of all ground fluents is the union of all reachable states, while the minimal set of potential

actions is the union of the legal actions of all roles in all reachable states. However, computing the minimal sets $FDom$ and $ADom$ requires to enumerate all reachable states, in general. Since this is infeasible for any game of practical interest, we only compute supersets of the domains.

In Section 5.2.1, we presented a procedure to compute supersets domains of all arguments of all functions and predicate of a game, with the help of so-called domain graphs. While this method is suitable for deriving a set of ground moves and fluents, it often contains unnecessary terms in the domains. However, for the proving properties using ASP, it is important to compute supersets of the domains that are as small as possible in order to reduce the size of the grounded program.

Here, we present an improved version of the algorithm to compute domain information. Furthermore, we show how this domain information can be used to efficiently encode the answer set programs for proving state properties.

We compute the supersets of the domains of all relations and functions of the game description by generating an *improved domain graph* from the rules of the game description. These improved domain graphs differ from domain graphs from Section 5.2.1 only by the fact that they are directed graphs.

Consider again the following rules encoding a common step counter.

```
succ(0, 1).
succ(1, 2).
succ(2, 3).
init(step(0)).
next(step(X)) :-
    true(step(Y)),
    succ(Y, X)).
```

Figure 7.2 depicts the improved domain graph for these rules.

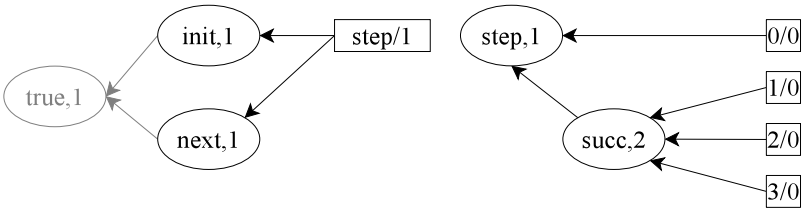

Figure 7.2.: An improved domain graph for calculating domains of functions and predicates. Ellipses denote individual arguments of functions or predicates while squares denote constants or function symbols.

Let us now formally define improved domain graphs.

Definition 7.5 (Improved Domain Graph)**.** *Let D be a GDL specification. Let D' be D together with the following three rules:*

```
    true(F)    :- init(F).
    true(F)    :- next(F).
```

```
does(R,M) :- legal(R,M).
```

An improved domain graph for a GDL specification D is the smallest directed graph $D = (V, E)$ with vertices V and edges E such that:

- *For every n-ary predicate or function p in D', $p/n \in E$, $(p, i) \in V$ for all $i \in \{1, \ldots, n\}$.*
- *If a constant c occurs as i-th argument of a predicate or function p in the head of a rule in D', then $c \to (p, i) \in E$.*
- *If a function $f(x_1, \ldots, x_n)$ occurs as i-th argument of a predicate or function p in the head of a rule in D', then $f/n \to (p, i) \in E$.*
- *If a variable occurs as i-th argument of a predicate or function p in the head of a rule $r \in D'$ and as j-th argument of a predicate or function q in a positive literal in the body of r, then $(q, j) \to (p, i) \in E$.*

Informally speaking, there is a node in the graph for every argument position of each function symbol and predicate symbol in the game description. For example, Figure 7.2 contains the nodes $(step, 1)$ and $(succ, 2)$ referring to the first argument of the function *step* and the second argument of the binary predicate *succ*, respectively. Furthermore, there is a node for each constant and function symbol. For example, the nodes 0/0, 1/0, 2/0, and 3/0 for the constants 0, 1, 2, and 3 and the node $step/1$ for the unary function *step*. There is an edge between an argument node and a constant (or function symbol) node if there is a head of a rule in the game description where the constant (or function symbol) appears in the respective argument of a function or predicate. For example, there is an edge between the nodes $step/1$ and $(init, 1)$ because of the rule `init(step(0)).`. Furthermore, there is an edge between two argument position nodes if there is a rule in the game in which the same variable appears in both arguments, once in the head and once in the body of the rule. For example, because of shared variable `X` in the rule `next(step(X)) :- true(step(Y)), succ(Y, X)).`, there is the edge $(succ, 2) \to (step, 1)$. The three additional rules capture the intuition that a fluent in the game is either true initially or in some successor state and that any legal move may potentially be executed by some player.

After constructing the domain graph $D = (V, E)$ from the game rules, we compute its transitive closure $D^+ = (V, E^+)$. The domains of all predicates and functions in the game description can now be defined as follows.

Definition 7.6 (Improved Domain)**.** *Let $D^+ = (V, E^+)$ be the transitive closure of the domain graph for a GDL specification D. Let $dom(p/n)$ denote the set of all ground instances of the n-ary predicate or function p and $dom(p, i)$ denote the domain of the i-th argument of predicate or function p:*

- $dom(p, i) = \{c : c/0 \to (p, i) \in E^+\} \cup \bigcup_{q/n \to (p,i) \in E^+, n>0} dom(q/n)$
- $dom(p/n) = \{p(x_1, \ldots, x_n) : x_1 \in dom(p, 1), \ldots, x_n \in dom(p, n)\}$

Note that, the definition above yields finite sets for all domains if the extended GDL specification D' from Definition 7.5 obeys the recursion restriction (Definition 2.7, condition 3). Otherwise, that is, if D but not D' satisfies the recursion

restriction, the set of reachable states of the game and, thus, the set of ground fluents might be infinite. However, in this thesis we only deal with finite games, as stated in Section 2.2.3.

With the definitions above we can compute the set of ground fluents $FDom$ as the domain of the first (and only) argument of the predicate `true`, that is, $FDom = dom(\texttt{true}, 1)$.

This domain information can be encoded as an ASP program, as follows.

Definition 7.7 (ASP Encoding of Domains)**.** *Let $D^+ = (V, E^+)$ be the transitive closure of the domain graph for the GDL specification D. Let $\eta(dom(v))$ be a predicate symbol which represents a unique name for the domain of $v \in V$, such that $\eta(dom(v))$ does not occur in D. The encoding Dom of the domains of all predicates and functions of D consists of the following ASP rules:*

- *For every vertex $p/n \in V$,*

$$\eta(\texttt{dom(p/n)})(\texttt{p}(\texttt{X}_1, \ldots, \texttt{X}_\texttt{n})) \texttt{ :- } \eta(\texttt{dom(p,1)})(\texttt{X}_1), \ldots, \eta(\texttt{dom(p,n)})(\texttt{X}_\texttt{n}).$$

- *For each edge $c/0 \to (p, i) \in E^+$,*

$$\eta(\texttt{dom(p,i)})(\texttt{c}).$$

- *For each edge $q/n \to (p, i) \in E^+$ with $n > 0$,*

$$\eta(\texttt{dom(p,i)})(\texttt{X}) \texttt{ :- } \eta(\texttt{dom(q/n)})(\texttt{X}).$$

It is easy to see that the (unique) answer set of Dom contains $\eta(dom(p, i))(t)$ for some term ground term t if, and only if, $t \in dom(p, i)$.

Using the above definition, we replace the state generator in the first line of (7.2) with the following rule:

$$0\,\{\,\texttt{true(F)} \texttt{ : } \eta(\texttt{dom(true,1)})(\texttt{F})\}.$$

In a similar fashion, we can obtain a straightforward encoding of an action generator (line 2 of (7.2)):

$$1\,\{\,\texttt{does(R,M)} \texttt{ : } \eta(\texttt{dom(does,2)})(\texttt{M})\}1 \texttt{ :- } \texttt{role(R)}.$$

However, this definition ignores the fact that different roles might have different potential actions and, thus, yields too many ground instances of actions for many games in practice. In other words, while usually all fluents in $dom(\texttt{true}, 1)$ may actually occur in a reachable state, many of the actions in $dom(\texttt{does}, 2)$ are never legal for any player. By Definition 7.6, domains of functions and predicates are essentially computed as the cross product of the domains of their arguments. Consider, then, a game like Checkers with the action `move(Piece,X1,Y1,X2,Y2)` of moving a piece from cell `X1,Y1` to cell `X2,Y2`.[1] In Checkers there

[1] See the repository `www.general-game-playing.de` for a complete encoding of Checkers.

are 4 different pieces ("men" and "kings" of either of two colours), and it is played on an 8 by 8 board. Thus, $dom(move/5)$ alone contains $4 * 8^4 = 16,384$ ground instances. However, due to the restrictions of how pieces move in Checkers, only a few hundred moves are actually possible. The problem becomes even more apparent with more complicated actions, e. g., `triplejump(Piece,X1,Y1, X2,Y2,X3,Y3,X4,Y4)`.

For these reasons, we encode the action generator (line 2 of (7.2)) differently. The idea is to compute a *static* version P^{legal}_{static} of the rules that define the legal moves of the players. Here, static means a relaxation of the rules that is independent of `true`, defined as follows.

Definition 7.8 (Static Legal Rules)**.**
Let D be a GDL specification and $\eta(static(p))$ be a predicate symbol which represents a unique name for the static version of predicate p. For each rule $p(\vec{X})$ `:-` B such that $p =$ `legal` or `legal` depends on p in the dependency graph of D with positive edges, P^{legal}_{static} contains the rule

$$\eta(static(p))(\vec{X}) \texttt{ :- } B_{static}.$$

where B_{static} comprises the following literals:

$$\{\ \eta(dom(\texttt{true},1))(\vec{Y})\ \mid\ true(\vec{Y}) \in B\ \} \cup$$
$$\{\ \eta(static(q))(\vec{Y})\ \mid\ q(\vec{Y}) \in B \wedge q \neq \texttt{true}\ \} \cup$$
$$\{\ \textbf{\texttt{not}}\ q(\vec{Y})\ \mid\ \textbf{\texttt{not}}\ q(\vec{Y}) \in B \wedge q \neq \texttt{true} \wedge q \textit{ does not depend on } \texttt{true}\ \}$$

Based on this definition, the action generator can be replaced by the clauses P^{legal}_{static} together with the following clause:

$$\texttt{1}\,\{\,\texttt{does(R,M)}\ :\ \eta(\texttt{static(legal)})(\texttt{R},\texttt{M})\}\ \texttt{1} :- \texttt{role(R)}.$$

7.4.3. Order of Proofs

As discussed in Section 7.1 above, some properties can only be proved with our proof method if other properties are already known to hold. For example, the property $\varphi_{\texttt{cell}}$ that $\{1, 2\}$ is a set of input arguments for the fluent `cell` in Tic-Tac-Toe can only be proved if it is known, that only one player is in control in every reachable state (property $\varphi_{\texttt{control}}$). Thus, attempting to prove $\varphi_{\texttt{cell}}$ before proving $\varphi_{\texttt{control}}$ results in a second proof attempt for $\varphi_{\texttt{cell}}$ according to Algorithm 5.

Ideally, we would identify these dependencies between the properties and attempt to prove a property φ_2 that depends on φ_1 only after proving φ_1. However, there is currently no easy way to identify these dependencies. Therefore, we use the following heuristics for the order in which properties are selected in line 4 of Algorithm 5: Select the property $\varphi \in \Phi_?$ first for which the splitting set U for the proof of the induction step is smallest. This heuristics is justified by the following facts:

- Typically, properties with a small splitting set, that is, a small set of literals that are relevant for the proof, have fewer dependencies on other properties.
- The size of the splitting set U determines the size of ASP $b_U(P)$ which has to be solved and small ASPs are typically solved faster. Furthermore, properties that are selected first tend to be attempted to be proved more often. Thus, selecting properties with small splitting sets tends to reduce the overall runtime of the algorithm.

7.4.4. Reducing the Number of Properties to Prove

Of course, the overall runtime of the proof procedure can reduced by reducing the number of properties that are attempted to be proved. One way, by which we reduce the number of properties is that we remove all properties that definitely do not hold. That is, we remove all properties φ for which a counter example exists, i. e., a reachable state that violates φ. Observe that in general the proof method is correct but incomplete. That is, if there is neither an answer set for the base case nor for the induction step then the property holds. The reverse implication does not hold: An answer set for the induction step does not imply the existence of a reachable state that violates φ. However, an answer set for the base case of the proof of property φ coincides with the fact that s_0 violates φ, i. e., φ does not hold in the initial state of the game. Thus, the existence of an answer set for the base case means that the initial state s_0 of the game is a counter example for the property. Hence, when the base case proof of a property fails we remove the property from Φ_{all} and do not attempt to prove it again.

Another reduction stems from the fact that some of the properties that we try to prove are stronger than others. For example, if $\emptyset$ is a set of input arguments for `control(X)` so is $\{1\}$: if there is just one `control` fluent in every state then there is also at most one for each role. More general, if I is a set of input arguments for some fluent, so is every set I' with $I \subseteq I'$. That is, every set of argument indices larger than I is a set of input arguments, too. Thus, when we have successfully proved that I is a set of input argument for some fluent f, we can immediately add all state properties that refer to larger sets of input arguments for f to the set of proved properties Φ. In order to maximise this effect, we also change the order in which the properties are selected such that the property "I is a set of input arguments for fluent f" is selected before "I' is a set of input arguments for fluent f" if $I \subset I'$.

7.5. Experimental Results

We conducted experiments of our system with a wide range of games from previous GGP competitions. We selected 12 representative games for the following presentation.

Two different experiments were run. The goal of the first experiment (Figure 7.3) was to show the effectiveness of the proof method. For this experiment we

manually identified boards and control fluents in the 12 games, that is, the input arguments of the fluents that encode boards and control in those games. For example, the property "board" in Tic-Tac-Toe refers to the property that $I = \{1, 2\}$ is a set of input arguments of fluent *cell*. We ran the proof method for the following properties:

control Proving that $\emptyset$ is a set of input arguments for the control fluent, that is the argument of the control fluent is unique in every state. The entry in the table is "n/a" if a game has no control fluent.

board Proving that the content of a board's cell is unique. That is, we try to prove that the arguments of a cell that describe coordinates of a board are input arguments.

board given control Proving that the content of a board's cell is unique given the information that the "control" property holds. Again, the entry in the table is "n/a" if a game has no control fluent.

zerosum Proving that the game is a zerosum game, that is, in every terminal state the rewards for all players add up to the same constant sum.

	control	board	board given control	zerosum
3pttc	(yes,0.00)	(no,0.24)	(no,0.28)	(no,0.00)
8puzzle	n/a	(no,0.14)	n/a	(no,0.00)
amazons	(yes,0.00)	(-,70.18)	(-,70.08)	(yes,0.38)
blocker	n/a	(yes,0.00)	n/a	(yes,0.00)
checkers	(yes,0.02)	(-,25.96)	(-,25.94)	(yes,1.14)
connectfour	(yes,0.00)	(no,0.01)	(yes,0.01)	(yes,0.01)
endgame	(yes,0.02)	(-,61.50)	(-,61.37)	(yes,0.12)
knightthrough	(yes,0.00)	(no,8.34)	(yes,51.36)	(yes,0.00)
othello	(yes,0.12)	(-,60.01)	(-,61.13)	(-,57.03)
pacman3p	(yes,0.01)	(no,0.33)	(no,0.27)	(no,0.09)
quarto	n/a	(no,2.36)	n/a	(yes,3.14)
tictactoe	(yes,0.00)	(no,0.00)	(yes,0.00)	(yes,0.00)
tttcc4	(yes,0.01)	(-,64.00)	(-,64.00)	(no,0.01)

Figure 7.3.: Results of proving some hand selected properties and runtimes of the proof procedure.

All experiments were run on an Intel Core 2 Duo CPU with 3.16GHz. A "-" means the prover was aborted because it used more than 1 GB of RAM. As can be seen from the results, proving some properties (e. g., control and zerosum) is very fast and successful for most of the games while proving other properties (e. g., board) is usually expensive and only possible in few games. The main influence on the time and space complexity is the number and size of the rules of the answer set program. We can see that our reduction technique from Section 7.4.1 is effective by comparing the times for control and board. The splitting set and therefore the ASP for the control property is typically much smaller than that of the board property. This is reflected in the lower runtime of the proofs for "control" compared to "board".

For some games, such as amazons, endgame, and othello, the ASP rules for

"board" are so complex that the answer set program cannot be successfully grounded because the answer set solver runs out of memory. Another reason why many properties that actually hold cannot be proved is that they can only be proved at the same time with some other properties. This happens if the properties are interdependent. Changing the algorithm to accommodate for interdependent properties should be straightforward. However, in the worst case an exponential number of combinations of properties have to be considered.

In the second experiment we ran our automatic proof method (Algorithm 5) as presented in Section 7.3 with all Φ_{all} being the set of all possible properties for input arguments, that is, for any combination of arguments of every fluent of the game the property that this combination of arguments is a set of input arguments. The results depicted in Figure 7.4 show that Algorithm 5 terminates in a reasonable amount of time for all games. Thus, it is possible to use it for detecting properties of games during the start clock of a match.

	all input arguments
3pttc	1.99
8puzzle	4.66
amazons	491.36
blocker	0.11
checkers	256.07
connectfour	0.14
endgame	433.83
knightthrough	56.02
othello	422.0
pacman3p	3.05
quarto	16.52
tictactoe	0.13
tttcc4	447.65

Figure 7.4.: Runtime in seconds to (attempt) to prove all possible input arguments of all fluents for a selection of games.

7.6. Summary and Outlook

The ability to prove properties of hitherto unknown games is a core ability of a successful general game playing system. We have shown that Answer Set Programming provides a theoretically grounded and practically feasible approach to this challenge, which not only is more reliable but often even faster than making informed guesses based on random sample matches. On the other hand, our experiments have also shown that state-of-the-art ASP systems cannot always be applied to prove properties of complex games in time reasonable for practical play. A promising alternative approach to tackle these games is given by the very recently developed method of first-order Answer Set Programming [Lee and Meng, 2008], by which grounding is avoided. A major challenge for

future work is to develop implementation techniques for first-order ASP systems and apply it to GGP.

One restriction of our approach is that we can only prove properties that can be described as a state property, i. e., a property referring to a single state. There are other interesting properties that one might want to prove, for example, the persistence of a fluent:

A fluent f is true (false) persistent if it stays true (false) once it is true (false). For example, `cell(a,1,x)` in Tic-Tac-Toe is true persistent, while `cell(a,1,b)` is false persistent. Persistence of a fluent f can be formulated as "if $(\neg)f$ holds in a state s it also holds in every successor state of s". This sentence cannot be formulated as a state property because it refers to more than one state.

Persistence can be used in the state evaluation function (Chapter 5): If a fluent in a subgoal of the game is false currently and false persistent we can evaluate it with 0; if it is true and true persistent we can evaluate it with 1. A form of false persistence was already presented in Chapter 6: fluents that cannot be reached, i. e., where the distance $\delta(s, f) = \infty$. However, true persistence cannot be detected using the fluent graphs from Chapter 6 and is a valuable addition to the evaluation function.

In [TV10], the approach that we developed here was extended to properties formulated in temporal logic such that properties like persistence of fluents can be formulated and proved.

8. Symmetry Detection

Exploiting symmetries of the underlying domain is an important optimisation technique for all kinds of search algorithms. Typically, symmetries increase the search space and, thus, the cost for finding a solution to a search problem exponentially. There is a lot of research on symmetry breaking in domains such as CSP [Pug05], Planning [FL99], and SAT-solving [ARMS02]. However, the methods developed in these domains are either limited in the types of symmetries that are handled or are hard to adapt to the General Game Playing domain because of significant differences in the structure of the problem. To exploit symmetries in a general game playing domain, the system must be able to automatically detect symmetries based on the rules of the game.

We present an approach to transform the rules of a game into a vertex-labelled graph such that automorphisms of the graph correspond with symmetries of the game and prove that the approach is sound. The algorithm detects many kinds of symmetries that often occur in games, e. g., rotation and reflection symmetries of boards, interchangeable objects, and symmetric roles. Furthermore, we present an extension for search algorithms that exploits the symmetries to prune the search space. Result presented in this chapter are published in [Sch10].

8.1. Games and Symmetries

Games in the general game playing domain are modelled as finite state machines as defined in Definition 2.1. A state of the state machine is a state of the game and actions of the players correspond to transitions of the state machine.

Several kinds of symmetries may be present in such a game, e. g., symmetries of states, moves, roles, and sequences of moves. Intuitively, symmetries of a game can be understood as mappings between objects such that the structure of the game is preserved. For example, two states of a game are symmetric if

- the same actions (or symmetric ones) are legal in both states,
- either both states or none of them is a terminal state,
- for each role both states have the same goal value, and
- executing symmetric joint actions in both states yields symmetric successor states.

Formally, we define a symmetry of a game as a mapping between states, actions and roles of the game:

Definition 8.1 (Symmetry)**.** *Let $\Gamma = (R, s_0, T, l, u, g)$ be a game over a set of ground terms Σ (Definition 2.2). A mapping $\sigma : \Sigma \to \Sigma$ is a symmetry of the game Γ if and only if the following conditions hold*

- $r \in R \equiv \sigma(r) \in R$
- $(\forall r, a, s)\ \ l(r, a, s) \equiv l(\sigma(r), \sigma(a), \sigma^s(s))$
- $(\forall A, s)\ \ u(\sigma^a(A), \sigma^s(s)) = \sigma^s(u(A, s))$
- $s \in t \equiv \sigma^s(s) \in t$
- $(\forall r, s)\ \ g(\sigma(r), \sigma^s(s)) = g(r, s)$

The macros σ^s and σ^a map states and joint actions to symmetric states and joint actions, respectively, according to the following definition:

$$\begin{aligned} \sigma^s(s) &\stackrel{\text{def}}{=} \{\sigma(x) | x \in s\} \\ \sigma^a(A) &\stackrel{\text{def}}{=} A' \text{ such that } A'(\sigma(r)) = \sigma(A(r)) \text{ for all } r \in R \end{aligned}$$

We will omit the superscripts on σ^s and σ^a in the rest of the thesis and denote both with σ.

A symmetry of a game expresses role, state, and action symmetries at the same time. For example, the state s in Figure 8.1 can be mapped to the state $\sigma(s)$ by

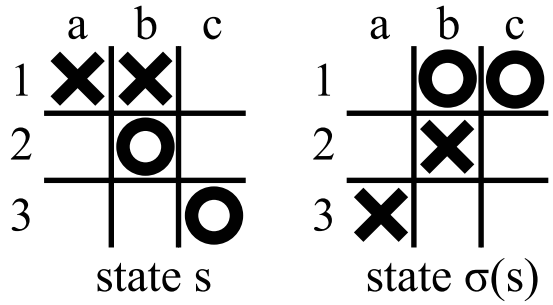

Figure 8.1.: Two states of Tic-Tac-Toe that are symmetric if the symbols `x` and `o` are swapped.

a symmetry σ. This symmetry σ exchanges the x-coordinates a and c ($\sigma(\mathtt{a}) = \mathtt{c}$, $\sigma(\mathtt{c}) = \mathtt{a}$), i. e., it reflects the board on the y-axis. However, σ also exchanges the symbols `x` and `o` and therefore the roles `xplayer` and `oplayer`. Thus, the two states are symmetric only if the two roles of the game are swapped. This is not what one would expect the term "symmetric states" to mean. Therefore, we give a more intuitive definition for symmetric states here.

Definition 8.2 (Symmetric States)**.** *Let $\Gamma = (R, s_0, T, l, u, g)$ be a game. A symmetry σ of game Γ is a state symmetry of Γ iff $(\forall r \in R)\sigma(r) = r$. Two states $s_1, s_2 \in \mathcal{S}$ are called symmetric if there is a state symmetry σ with $\sigma(s_1) = s_2$.*

A state symmetry is a special case of a symmetry of a game. According to the definition, a state symmetry maps each role of the game to itself. Two states are symmetric if there is a symmetry mapping one state to the other without

affecting the roles. Thus, the two states from Figure 8.1 are not symmetric according to this definition.

Since the result of a joint action depends on the state it is applied in, it is only meaningful to define symmetric actions with respect to one particular state:

Definition 8.3 (Symmetric Actions)**.** *Let $\Gamma = (R, s_0, T, l, u, g)$ be a game with states $\mathcal{S}$ and actions $\mathcal{A}$. Two joint actions $A_1, A_2 : R \to \mathcal{A}$ are called symmetric in a state $s \in \mathcal{S}$ if and only if there is a state symmetry σ of Γ with $\sigma(s) = s$ and $\sigma(A_1) = A_2$.*

From the definition of symmetry (Definition 8.1) it follows that the states resulting from the execution of two symmetric joint actions are symmetric:

Proposition 8.1. *Let σ be a state symmetry of a game $\Gamma = (R, s_0, T, l, u, g)$ and $A_1, A_2 \in \mathcal{A}$ be joint actions of Γ. If $\sigma(A_1) = A_2$ then $\sigma(u(A_1, s)) = u(A_2, s)$ for all states s of Γ.*

Proof. $\sigma(u(A_1, s)) = u(\sigma(A_1), \sigma(s))$ according to Definition 8.1. Since σ is a state symmetry, $\sigma(s) = s$. Furthermore, by assumption $\sigma(A_1) = A_2$. Hence, $\sigma(u(A_1, s)) = u(\sigma(A_1), \sigma(s)) = u(A_2, s)$ □

Note that the symmetries of a game are independent of the initial state of the game. As a consequence, all games that only differ in the initial state have the same set of symmetries.

Although possible in principle, using the state machine model of a game to compute symmetries is not feasible for all but the easiest of games because of the size of the state machine. Luckily, games are not described as state machines directly, but in the form of modular rules (see Section 2.2). The rules of a game are typically exponentially smaller than the state machine they represent. We want to take advantage of this compact representation of games and find symmetries of a game by analysing the rules of the game instead of the game itself. For this purpose we transform the rules of the game, which are typically given in the Game Description Language (GDL), into a vertex labelled graph and compute automorphisms of the graph in order to find symmetries of the game.

8.2. Rule Graphs

It is not a new idea to use graph automorphisms to compute symmetries of a problem. This approach has been successfully applied to constraint satisfaction problems [Pug05] and SAT solving [ARMS02], among others. However, a key for using this method is to have a graph representation of the problem such that the graph has the same symmetries.

Unrelated to symmetries in games, Kuhlmann et. al. describe a mapping of GDL game descriptions to so called “rule graphs” such that two rule graphs are

isomorphic if and only if the game descriptions are identical up to renaming of non-keyword constants and variables [KS07]. Basically, rule graphs contain vertices for all predicates, functions, constants, and variables in the game description and connections between these vertices that match the structure of the rules. The nodes of rule graphs are labelled such that isomorphisms can only map constants to other constants, variables to variables, etc.

We argue that these graphs can be used to compute symmetries of games. If there is an automorphism of such a rule graph, that means an isomorphism of the graph to itself, then there is a scrambling of the game description that does not change the rules of the game. Since constants of the game description refer to objects in the game, a mapping between constants that does not change the rules describes configurations of objects that are interchangeable in the game.

For example, it can easily be seen that consistently interchanging the constants `a` and `c` (or `1` and `3`) in the rules of Tic-Tac-Toe (Figure 2.1) yields the same set of rules which means that the "objects" referred to by these constants are interchangeable. In this example the objects stand for coordinates of a board, swapping of `a` and `c` or `1` and `3` corresponds to horizontal or vertical reflection of the board, respectively. To illustrate that the mapping will indeed result in the same set of rules, consider the following rule of Tic-Tac-Toe:

```
1  line(P) :- true(cell(a,1,P)),
2    true(cell(b,2,P)), true(cell(c,3,P)).
```

By swapping of `1` and `3` we obtain the rule

```
1  line(P) :- true(cell(a,3,P)),
2    true(cell(b,2,P)), true(cell(c,1,P)).
```

This rule is another rule of the Tic-Tac-Toe game. The same argument holds for all other rules of the game.

However, rotation symmetry of the board cannot be expressed by a mapping between constants of the game if the typical representation of a board, `cell(X,Y,_)`, is used. A rotation of the board would correspond to a suitable mapping between the coordinates plus the swapping of the row and column argument of the `cell` fluent. For example, if σ represent a clockwise rotation of 90 degrees, we have the following mapping from the fluents of state s of Figure 8.2 to fluents of the symmetric state $\sigma(s)$:

fluent f	$\sigma(f)$
`cell(a,1,x)`	`cell(c,1,x)`
`cell(b,1,x)`	`cell(c,2,x)`
`cell(c,1,b)`	`cell(c,3,b)`
`cell(a,2,b)`	`cell(b,1,b)`
`cell(b,2,o)`	`cell(b,2,o)`
`cell(c,2,b)`	`cell(b,3,b)`
`cell(a,3,b)`	`cell(a,1,b)`
`cell(b,3,b)`	`cell(a,2,b)`
`cell(c,3,o)`	`cell(a,3,o)`

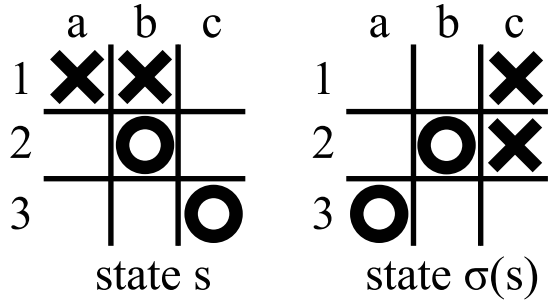

Figure 8.2.: Clockwise rotation of a Tic-Tac-Toe board.

It can be seen that the mapping of constants $a \mapsto 1$, $b \mapsto 2$, $c \mapsto 3$, $1 \mapsto c$, $2 \mapsto b$, and $3 \mapsto a$ together with exchanging the first two arguments of the `cell` fluent results in the mapping above.

The rule graphs from [KS07] do not allow the exchange of arguments. Therefore, we propose enhanced rule graphs, which differ from the rule graphs from [KS07] mainly by replacing the ordering edges between arguments with argument index vertices.

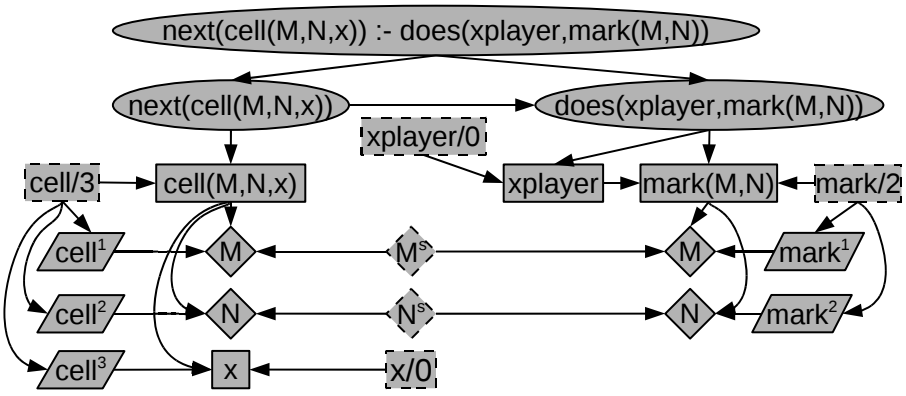

Figure 8.3.: The enhanced rule graph for the rule **`next`**`(cell(M,N,x)) :-` **`does`**`(xplayer,mark(M,N))` with labelled nodes for illustration.

In Figure 8.3 you can see the enhanced rule graph for one of the rules of Tic-Tac-Toe. Different labels of the nodes are depicted by different shapes and borders. The labels that are shown in the nodes are only there for illustration. The actual graph as used for symmetry detection is shown in Figure 8.4.

The graph contains one node for every part of the rule. For example, the node on top refers to the rule as a whole, and the node **`next`**`(...)` refers to the head of the rule. Nodes referring to rules are connected to the nodes referring to the head and to literals in the body of the rule. Likewise, nodes referring to predicates or functions are connected to the nodes that refer to the terms that are arguments

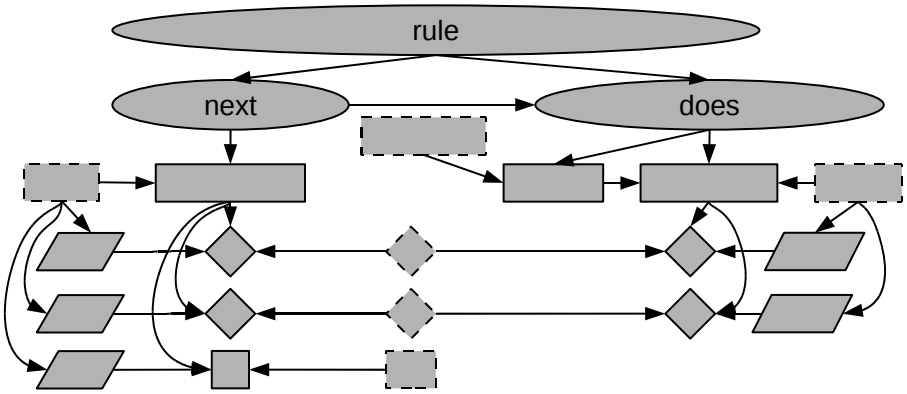

Figure 8.4.: The enhanced rule graph for the rule `next(cell(M,N,x)) :- does(xplayer,mark(M,N))`.

of these predicates or functions, e. g., $next(cell(M, M, x)) \rightarrow cell(M, N, x)$. In addition, there are nodes with a dashed border that refer to symbols of the game description. For example, `cell/3` refers to the ternary function `cell` that represents the Tic-Tac-Toe board and $\texttt{M}^{\texttt{s}}$ refers to the variable name `M` that is used in the rule. These symbol nodes are connected to the other nodes in the graph in which the symbol occurs. Furthermore, the nodes in the shape of a parallelogram, such as $\texttt{cell}^1$ or $mark^2$, represent argument indices of the respective functions. These nodes are connected to the nodes that represent the actual arguments of the functions as they occur in the rule, e. g., $mark^2$ is connected to N because the variable N occurs in the second argument of the function $mark$ in the rule.

The argument index nodes are the nodes that distinguish enhanced rules graphs from the rule graphs in [KS07]. These nodes enable us to find automorphisms (and, thus, symmetries) that interchange arguments of functions and predicates. An automorphism of a graph is a mapping between vertices of the graph such that the structure of the graph is preserved. It can intuitively be seen in Figure 8.3 that there is the automorphism $\sigma = \{cell^1 \mapsto cell^2, cell^2 \mapsto cell^1, M \mapsto N, N \mapsto M, M^s \mapsto N^s, N^s \mapsto M^s, mark^1 \mapsto mark^2, mark^2 \mapsto mark^1\}$. This mapping simultaneously interchanges the first two arguments of $cell$ and of $mark$ as well as the variables M and N. The (non-enhanced) rule graph for the same rule of Tic-Tac-Toe as shown in Figure 8.5 does not allow this mapping due to its use of ordering edges between the arguments of functions instead of argument index nodes.

Formally, we define enhanced rule graphs as follows:

Definition 8.4 (Enhanced Rule Graph)**.** *Let D be a valid GDL game description. The enhanced rule graph of D is the smallest vertex labelled graph $G = (V, E, l)$ with the following properties:*

- *For every n-ary non-keyword relation symbol or function symbol p^1 in D*

[1] We treat constants as null-ary functions.

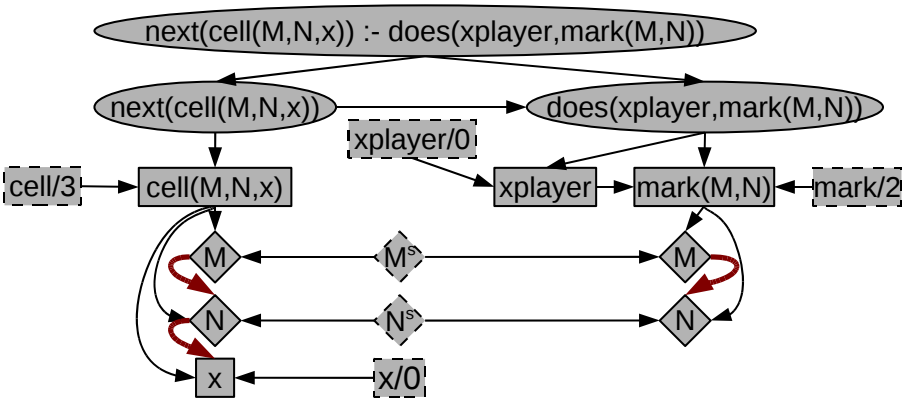

Figure 8.5.: The rule graph for the rule `next(cell(M,N,x)) :- does(xplayer,mark(M,N))` according to [KS07]. Unlike the enhanced rule graph in Figure 8.3, this graph does not exhibit a symmetry between M^s and N^s.

that is not a goal value[2] *and for all* $i \in [1, n]$

- $p/n \in V$ *(stands for relation or function symbol* p *with arity* n*),*
- $p^i \in V$ *(stands for the* i*-th argument index of* p*),*
- $(p/n, p^i) \in E$*,*
- $l(p/n) = \texttt{symbol}_{\texttt{const}}$*, and*
- $l(p^i) = \texttt{arg}$*.*

- *For every goal value* g *in* D
  - $g/0 \in V$*, and*
  - $l(g/0) = g$ *(to ensure that goal values are not mapped to each other).*
- *For every variable symbol* v *in* D
  - $v^s \in V$ *and*
  - $l(v^s) = \texttt{symbol}_{\texttt{var}}$*.*

Furthermore, for every part v *of* D

- *If* $v = \texttt{h} :- \texttt{b}_1, \ldots, \texttt{b}_\texttt{n}$ *is a rule then*
  - $v \in V$*,*
  - $(v, h), (v, b_1), \ldots (v, b_n) \in E$*,*
  - $(h, b_1), \ldots, (h, b_n) \in E$ *(in order to make a distinction between head and body of a rule), and*
  - $l(v) = \texttt{rule}$*.*
- *If* $v = \texttt{not a}$ *is a negative literal then*
  - $v \in V$*,*
  - $(v, a) \in E$*, and*

[2] Goal values are integers that might occur as the goal value $g(r, s)$ of some role r in some state s of the game. They can be identified by computing the domain of the second argument of `goal(R,V)` with the algorithms described in Sections 5.2.1 and 7.4.2.

  - $l(v) = \text{not}$.

- *If* $v = \texttt{distinct}(\texttt{t}_1, \texttt{t}_2)$ *then*
  - $v \in V$,
  - $(v, t_1), (v, t_2) \in E$, *and*
  - $l(v) = \texttt{distinct}$.
- *If* $v = \texttt{p}(\texttt{t}_1, \ldots, \texttt{t}_\texttt{n})$ *is an atom and* `p` *is a keyword (*`true`, `does`, `legal`, *...) other than* `distinct` *then*
  - $v \in V$,
  - $(v, t_1), \ldots, (v, t_n) \in E$,
  - $(t_1, t_2), \ldots, (t_{n-1}, t_n) \in E$ *(to define the order of the arguments of* v*), and*
  - $l(v) = \texttt{p}$.
- *If* $v = \texttt{p}(\texttt{t}_1, \ldots, \texttt{t}_\texttt{n})$ *is an atom and* p *is not a GDL keyword then*
  - $v \in V$,
  - $(p/n, v) \in E$ *(to relate predicate symbol node* p/n *introduced above with* v*),*
  - $(v, t_1), \ldots, (v, t_n) \in E$,
  - $(p^1, t_1), \ldots, (p^n, t_n) \in E$ *(to relate argument indices of* p *with the actual arguments of* $\texttt{p}(\texttt{t}_1, \ldots, \texttt{t}_\texttt{n})$*), and*
  - $l(v) = \texttt{predicate}$,
- *If* $v = \texttt{f}(\texttt{t}_1, \ldots, \texttt{t}_\texttt{n})$ *is a function then*
  - $v \in V$,
  - $(f/n, v) \in E$, *(to relate function symbol node* f/n *introduced above with* v*)*
  - $(v, t_1), \ldots, (v, t_n) \in E$,
  - $(f^1, t_1), \ldots, (f^n, t_n) \in E$ *(to relate argument indices of* f *with the actual arguments of* $\texttt{f}(\texttt{t}_1, \ldots, \texttt{t}_\texttt{n})$*), and*
  - $l(v) = \texttt{function}$.
- *If* v *is a variable then*
  - $v \in V$,
  - $(v^s, v) \in E$, *and*
  - $l(v) = \texttt{variable}$.

In this definition, we consider only game descriptions where variables in different clauses are named differently. Every game description can be easily transformed into an equivalent one which meets this requirement. Note that every occurrence of an atom or term is treated as a different atom or term. That means if the same term occurs twice in the rules there are two vertices, one for each occurrence. This can be observed in Figure 8.3, which contains one node for each occurrence of the variables `M` and `N`. The different occurrences of an atom or term are however linked via the nodes for predicate symbols (p/n), function symbols (f/n) and variable symbols (v^s). There is just one symbol node for every symbol in the game description.

In the remainder of the chapter we write "rule graph" instead of "enhanced rule graph". All results apply to enhanced rule graphs.

8.3. Theoretic Results

Our first theorem describes the connection between automorphisms of rule graphs and scramblings of game descriptions. In order to reflect the reordering of arguments we extended the definition of a scrambling of a game description from [KS07] as follows:

Definition 8.5 (Scrambling of a Game Description)**.** *A scrambling of a game description D is a one-to-one function over function symbols, relation symbols, variable symbols, and argument indices of function symbols and non-keyword relation symbols in D.*

The first theorem says that there is a mapping between automorphisms of the rule graph and scramblings of a game description.

Theorem 8.1 (Scramblings and Automorphisms)**.** *Let D be a game description, $G = (V, E, l)$ be its rule graph and H be the set of automorphisms of G. We call two automorphisms h_1 and h_2 equivalent ($h_1 \sim h_2$) if they agree on the mapping of all symbol vertices and argument index vertices:*

$$h_1 \sim h_2 \stackrel{\text{def}}{=} l(v) \in \{\texttt{symbol}_{\texttt{const}}, \texttt{symbol}_{\texttt{var}}, \texttt{arg}\} \supset h_1(v) = h_2(v)$$

There is a one-to-one mapping between the quotient set $H/\sim$ and scramblings that map the game description D to itself.

Proof. Let M_G be the set of mappings over nodes from a rule graph G. The rule graph construction algorithm adds exactly one symbol label vertex to the graph for each symbol in the game description D, and exactly one argument index vertex for each argument index of all function symbols and non-keyword relation symbols in D. Thus, there are one-to-one mappings between symbols of D and symbol vertices (p/n, f/n, v^s) of the rule graph, and between argument indices of D and argument index vertices (p^i, f^i) of the rule graph. Hence, for every set $M \in M_G/\sim$ of mappings over nodes of the rule graph that coincide in the mapping of vertices v with labels $l(v) \in \{\texttt{symbol}_{\texttt{const}}, \texttt{symbol}_{\texttt{var}}, \texttt{arg}\}$ there is a scrambling m of the game description D and vice versa.

It remains to be proved that, ($\Rightarrow$) if $h \in M_G$ is an automorphism, then the associated scrambling m maps D to itself, and the reverse direction ($\Leftarrow$): if m maps D to itself then there is at least one automorphism of G in the set $M \in M_G/\sim$ that is associated to m.

We start with $\Leftarrow$: In general, a scrambling m maps a game description D to a game description $D' = m(D)$. The construction of a rule graph is deterministic and is independent on the names of predicates and functions and the argument positions of terms. Hence, the rule graphs G and G' of D and D', respectively, are

isomorphic. Since, $m(D) = D$ the rule graph $G' = G$. Thus, the isomorphisms between G' and G is an automorphisms of G.

The proof of the other direction ($\Rightarrow$) is identical to the proof in [KS07]. □

The important implication of the theorem is that we can compute all scramblings that map a game description to itself by computing all automorphisms of its rule graph. We call a scrambling m corresponding to an automorphism h of a rule graph if m and h coincide in the mapping of all predicate symbols, variable symbols, function symbols, and argument indices.

Intuitively, a scrambling m defines a bijective mapping σ_m between arbitrary ground terms of D by the following inductive definition:

$$\begin{aligned} \sigma_m(c) &= m(c) \quad \text{for all constants } c \\ \sigma_m(f(t_1, \ldots, t_n)) &= m(f)(t'_1, \ldots t'_n), \text{ where} \\ & \quad t'_j = \sigma_m(t_i) \text{ with } m(f^i) = m(f)^j \end{aligned}$$

Thus, σ_m replaces every symbol f in a term by the symbol $m(f)$ as defined by the scrambling of D. Furthermore, σ_m reorders the arguments t_i of functions according to the mapping of the argument indices, where $m(f^i) = m(f)^j$ indicates that the i-th argument of a function with symbol f is mapped to the j-th argument of a function with symbol $m(f)$.

In the following theorem we establish a correspondence between symmetries of games and automorphisms of rule graphs using the scrambling of the game description that the rule graph and the game belongs to.

Theorem 8.2 (Symmetries and Automorphisms)**.** *Let Γ be the game for a game description D, h be an automorphism of the rule graph of D, and m be a scrambling of D corresponding to h. Then σ_m is a symmetry of Γ corresponding to h.*

Proof. The proof uses the construction of the game Γ from the game description D to show that σ_m satisfies the defining properties of a symmetry. For example, we have to prove that $s \in T \equiv \sigma_m(s) \in T$, where T is the set of terminal states of Γ. By the construction of a game from a game description (Definition 2.14), $\sigma_m(s) \in T$ is equivalent to $D \cup \{true(f) | f \in \sigma_m(s)\} \models terminal$. This is equivalent to $D \cup \sigma_m(\{true(f) | f \in s\}) \models terminal$ because *true* is a keyword and keywords are mapped to themselves by m. Now since $\sigma_m(D) = D$ and *terminal* is a keyword, this is equivalent to $D \cup s^{true} \models terminal$, which is the definition of $s \in T$. The remaining properties of a symmetry are proved in the same way. □

We can summarise our theoretic results with the following diagram:

Rule Graph		**Game Description**		**Game**
G	$\xleftarrow{\text{Def. 8.4}}$	D	$\xrightarrow{\text{Def. 2.14}}$	Γ
$\downarrow h$	$\overset{\text{Theorem 8.1}}{\Longrightarrow}$	$\downarrow m$	$\overset{\text{Theorem 8.2}}{\Longleftrightarrow}$	$\downarrow \sigma$
G		D		Γ

There is a correspondence between rule graphs and game descriptions given by the definition of rule graphs (Definition 8.4). Theorem 8.1 shows that therefore there is a correspondence between an automorphism h of a rule graph G (i. e., a mapping between vertices of G) and a scrambling m of a game description D, (i. e., a mapping between symbols in D). The is an associated game Γ for each game description D given by Definition 2.14. By using all these correspondences, Theorem 8.2 shows that the the scrambling m corresponding to an automorphism h defines a symmetry σ of the game, i. e., a mapping between roles states and actions of the game that are defined wrt. the ground terms of the game description. Thus, automorphisms h of the rule graph of the game descriptions can indeed be used to compute symmetries of the game.

If we use the rule graph of the complete game description D to compute symmetries, we only get symmetries that are present in the initial state of the game, that is, symmetries σ with $\sigma(s_0) = s_0$. However, this is not required by the definition of symmetries (Definition 8.1). In some games there may be so called "dynamic symmetries", i. e., symmetries that occur only in some states of the game but are not present in the initial state. To also find automorphisms corresponding to these symmetries, we use the rule graph of $D' = D \setminus \{init(F) \in D\}$, i. e., the rules of D except for the initial state description. Observe that D can contain function symbols or constants that are not included in D'. If so, these symbols are only part of the initial state description and do not occur anywhere else in the rules of the game. Therefore, they refer to objects of the game that are interchangeable and can be arbitrarily mapped to each other by the symmetry.

With minor changes, the approach can be used for computing only certain types of symmetries. For example, to compute only state symmetries (cf. Definition 8.2), we can assigning each node belonging to a `role`-fact a different label. Symmetries of a particular state s, i. e., symmetries with $\sigma(s) = s$, can be computed by using the rule graph of $D' \cup s^{true}$, where s^{true} is a set of clauses encoding state s as the current state of the game.

8.4. Exploiting Symmetries

Standard tools, like nauty (`http://cs.anu.edu.au/ ~bdm/nauty/`), are able to compute the automorphisms of a rule graph and, thus, the symmetries of a game efficiently. Even for large games, computing all automorphism of the rule graph takes at most a few seconds. This leaves the question of how to exploit the symmetries to improve game play. Depending on the approach used in the general game player, symmetries may be used in different ways, for example, to speed up analysation of the game's properties or to prune the search space. For example, proofs of game properties (see Chapter 7) can be made more efficient by skipping proofs of symmetric game properties or by using symmetry breaking techniques to reduce the set of reachable states that has to be analysed. Because all current general game players employ some kind of search to play general games, we present a way to use the symmetries for pruning the search space.

One way of pruning the search space is to prune symmetric joint actions in node

expansion. Symmetric joint actions lead to symmetric states in the game tree (cf. Proposition 8.1) and, thus, to symmetric subtrees of the game tree. Therefore, it it is sufficient to use only one joint action of each set of symmetric joint actions in a state for node expansion. However, this does not use the full potential of the available information. In particular, there may be two non-symmetric sequences of joint actions leading to symmetric states. The expansion of the second state is not avoided since the action sequences are not symmetric. For instance, the two action sequences of Tic-Tac-Toe depicted in Figure 8.6 are non-symmetric ($\not\simeq$) but lead to symmetric states ($\simeq$).

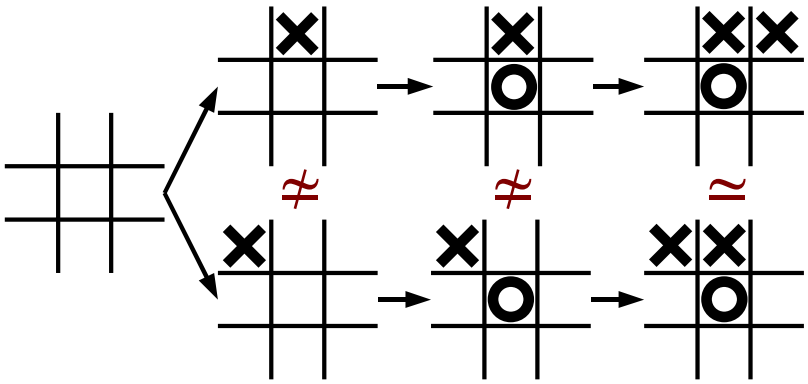

Figure 8.6.: Two non-symmetric actions sequences leading to symmetric states.

A common reason for this are transpositions of the action sequence. For example, in our formulation of Tic-Tac-Toe, the order in which the actions are executed is unimportant for the resulting state. Therefore, every transposition of a symmetric action sequence also leads to a symmetric state.

We propose to use a transposition table to detect those symmetric states before expanding a node. That means before we evaluate or expand a state in the game tree we check whether this state or any state that is symmetric to this one has an entry in the transposition table. If so, we just use the value stored in the transposition table without expanding the state. It is clear that the algorithm does not use any additional memory compared to normal search. On the contrary, the transposition table may get smaller because symmetric states are not stored. However, the time for node expansion is increased by the time for computing the symmetric states and checking whether some symmetric state is in the transposition table.

Therefore, it is essential to be able to compute hash values of states and symmetric states efficiently. We use Zobrist hashing [Zob70] where each ground fluent is mapped to a randomly generated hash value and the hash value of a state is the bit-wise exclusive disjunction (xor) of the hash values of its fluents. For efficiently computing symmetric states all ground fluents are numbered consecutively and the symmetry mappings are tabulated for the fluents. In our implementation the time to compute all symmetric states for some state depends on the game and ranges from $\frac{1}{50}$ to 3 times the time for expanding a state for the 13 games

we tried. The absolute time depends on the size of the state and the number of symmetries in the game. However, the time costs are only relative to the time costs for expanding a state in the game tree, which depends on the complexity of the **`legal`** and **`next`** rules of the game.

We conducted experiments on a selection of games where we measured the time it took to do a depth-limited search in every state on a path through the game. We compared:

- normal search with a transposition table but without checking for symmetric states ("normal search"),
- the approach where only symmetric actions were pruned ("prune symmetric moves"), and
- the approach where we check all symmetric states before expanding a state ("check symmetric states").

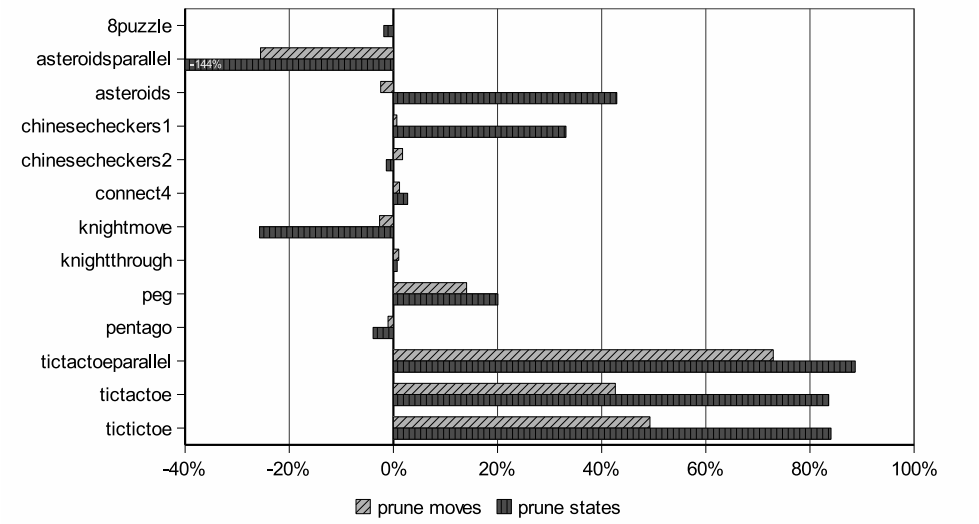

Figure 8.7.: The chart shows the time savings of using search with pruning symmetric moves and pruning symmetric states compared to normal search, i. e., without using any symmetry information.

In Figure 8.7 the time savings for search with symmetry pruning compared to "normal search" are shown. Table 8.1 shows runtimes of the "normal search" and the depth limits we used for the games.

It can be seen that for the majority of games exploiting the symmetries improves the performance. Also, in most cases the additional effort of transposition table look-up for all symmetric states pays off compared to pruning only symmetric moves. This is not too surprising because for pruning symmetric moves in a state we have to compute the symmetries that map the state to itself. In many cases this is only slightly faster then computing all symmetric states.

For some games the overhead of checking for symmetric states is higher than the gain, most notably asteroidsparallel, which is just two instances of asteroids played in parallel. The bad result has several reasons. One problem is that

Game	Depth	Runtime (in s)	# States	# Symmetries
8puzzle	15	81	389286	1
asteroidsparallel	7	375	1460896	127
asteroids	15	356	2575774	7
chinesecheckers1	11	24	63761	1
chinesecheckers2	7	66	287483	1
connect4	6	147	94372	1
knightmove	7	81	168849	7
knightthrough	4	1071	866625	1
peg	7	265	109154	7
pentago	5	312	661145	7
tictactoeparallel	4	80	117918	127
tictactoe	9	1.2	5478	7
tictictoe	9	8.3	12829	7

Table 8.1.: This table shows the depth-limit that we used for the depth-limited search procedure, the average runtime of the normal search, the number of states expanded by the normal search and the number of symmetries found in the game. Depth-limits were chosen such that the runtime of the search was still within the usual time limits of a match during a competition.

because of the rather large number of symmetries, computing all symmetric states is quite expensive. In knightmove the problem is that many symmetric states can only be reached after action sequences that are longer than the depth-limit. Additionally, because of the very simple rules of the game, computing state expansion is fast compared to computing symmetric states. For tictactoe and tictictoe the results are near optimal. Because every symmetric state is indeed reachable from the initial state and the complete game tree was searched about $\frac{7}{8} = 87.5\%$ of the states were not explored. The reason for the large number of symmetries in asteroidsparallel and tictactoeparallel is that these games consist of two independent instances of the same game. These games could be played much more efficiently by decomposing them [ZST09] and looking for symmetries in each subgame separately.

It should be noted that the experiments were run with blind search, i. e., without a heuristic evaluation of non-terminal leaf nodes, because we wanted to show the general applicability of our approach in a general game player regardless of the heuristics used. If heuristic search is used, the saved time is increased by the saved heuristic evaluation time, which may be considerable, depending on the complexity of the heuristic function. In our game player that means that even in games like 8puzzle and pentago exploiting symmetries pays off.

In order to avoid big negative impact like in asteroidsparallel or knightmove we keep track of the number n_{saved} of saved state expansions by counting the state expansions in each subtree during search and storing this number for each state in the transposition table. Whenever a symmetric state is found, we add the stored

number to n_{saved}. We estimate the saved time $t_{saved} = n_{saved} * t_{exp} - n_{total} * t_{sym}$, where t_{exp} is the average time for expanding a state, n_{total} is the total number of expanded states and t_{sym} is the average time for computing all symmetric states for a state. Both, t_{exp} and t_{sym} can be estimated by averaging over times measured for a number of random states of the game. If $t_{saved} < -t_{limit}$ for some predefined t_{limit}, we switch to normal search without symmetry exploitation thereby limiting the negative impact to t_{limit}.

8.5. Discussion

We presented a method that can be used to detect and exploit many symmetries that often occur in games, e. g., object symmetries (functionally equivalent objects), configuration symmetries (symmetries between collections of objects and their relations to each other), and action symmetries (actions leading to symmetric states). This includes the typical symmetries of board games, like rotation, and reflection, as well as symmetric roles.

None of the tested games contained object symmetries. This type of symmetries leads to a number of symmetries exponential in the number of functionally equivalent objects and should therefore be handled more efficiently than with our approach. The method described in [FL02] for planning can be easily adapted to the general game playing domain. Plan permutation symmetries, that are exploited in, e. g., [LF03], are not to be confused with our symmetric action sequences. Symmetric plan permutations are permutations of a plan that lead to the same state, whereas symmetric action sequences are sequences of element-wise symmetric joint actions. Plan permutation symmetries are typically exploited in a game playing program by a transposition table without any symmetry detection.

A previous approach to symmetry detection in general games is [BKS06]. The paper informally describes a method to detect certain symmetries in board games that is potentially very expensive because it requires to enumerate all states of a game. Furthermore, it only works under the assumption that one can detect the board in the game. Our approach is typically much cheaper because it is based on the game rules instead of the states, and more general because it is not limited to board games.

Because the symmetry detection is based on the game description instead of the game graph itself, it can only detect symmetries that are apparent in the game description. Consequently, symmetry detection based on different game descriptions for the same game may lead to different results. For example, consider the rules of Tic-Tac-Toe together with the tautological rule `p(a) :- distinct(a,a).` Adding this rule prevents the symmetry between `a` and `c` from being detected because interchanging `a` and `c` results in a different game description. Consequently, our approach may benefit from removing superfluous rules and transforming the game description to some normal form.

Another limitation of the approach is that it does not allow to map arbitrary terms to each other. For example, the approach cannot detect the symmetry in

a variant of Tic-Tac-Toe, where we rename `a` to `f(a)`, because an automorphism only maps single vertices to each other but `f(a)` is not represented by a single vertex in the rule graph, while `c` is. It is in principle possible to overcome this limitation by propositionalising a game description and using one vertex per proposition. The resulting rule graphs would be very similar to propositional automata and could in addition be used to improve reasoning speed [SLG08]. However, this is only feasible for small games because the ground representation of the game rules can be exponentially larger than the original one. Not only does propositionalising of large game descriptions take valuable time, but computing automorphisms of the resulting large rule graphs is also more expensive. A possible compromise is to partially ground the game rules in order to limit the size of the description but still benefit from the advantages of propositional representations when possible.

9. Related Work

In this chapter, we will give an overview of previous work on General Game Playing and related topics. We focus on the following topics that are important for knowledge-based general game players:

- search algorithms, and
- heuristics.

9.1. Search Algorithms

Search algorithms for games have been widely studied in the literature. However, most of the publications are concerned with specific games.

Only single-player games, as a subclass of general games have been studied in general, i. e., in a domain independent way. Multi-player games are typically more complex than single-player games because in single-player games there are no opponents whose objectives have to be taken into account. Therefore, we will treat search algorithms for single-player games separately in Section 9.1.1.

For multi-player games two fundamentally different kinds of search algorithms are used in General Game Playing today: heuristic search and Monte-Carlo simulation methods. Both kinds of algorithms are not specific to General Game Playing but are also used in classical computer game playing in different games. We will discuss heuristic search algorithms in Section 9.1.2 and Monte-Carlo simulation methods in Section 9.1.3.

9.1.1. Single-Player Games

Single-player games in GGP are conceptually equivalent to planning problems: There is only one agent and the goal is to manipulate the game state in such a way that a predefined condition (the goal) is fulfilled. Domain independent planning is a well-established research area.

The planning community uses PDDL [Mcd00] as a language to describe planning problems. PDDL is action-centred, i. e., for each action the effects are enumerated, while GDL specifies the conditions when each fluent holds in the successor state. This difference in representation makes it difficult to use algorithms and systems that were developed for domain independent planning in GGP to solve single-player games. Translating GDL to PDDL is non-trivial and may lead to an exponential blow-up in the size of the representation [KE09].

Apart from the difference in the representation language, systems for domain independent planning try to solve the whole problem up front. In GGP on the other hand, the start clock is typically limited such that solving the problem up-front is not possible. Instead, a GGP system still has time to find reasonably good moves, after the game started: there is still time to search for a solution during the play clocks of each step of the game. However, at that point the state of the game has already changed, thus the search algorithm has to incorporate this new information.

Attempts to directly apply planning systems to single-player games in GGP were not successful, so far, mainly because of these differences. Often, translating GDL descriptions to PDDL fails because of the exponential blow-up involved. In the other cases, the produced PDDL descriptions are hard to handle by current planning systems [Rüd09].

Current GGP systems are dealing with single-player games in different ways. As described in Section 4.4, we use a slightly modified iterative deepening depth-first search with a heuristic evaluation function for non-terminal states. The heuristic function that we use is the same for all classes of games. It is described in Chapter 5 and further refined in Chapter 6.

Cluneplayer [Clu08] uses various different heuristic search techniques for single-player games such as A* [RN95], minimum lookahead [Kor90], and depth-first search with 1-ply lookahead. He also runs uninformed search methods in parallel to the heuristic search. The latter methods may find a solution faster because they do not spend time on constructing and evaluating heuristics. However, his approach requires either independent parallel processes or a decision procedure that allocates a portion of the available time to each different search method. Both cases involve considerable computation overhead compared to running only a single search. Furthermore, deciding which search procedure works best for the game at hand or how much time should be allotted to each search procedure are open problems.

Other players use variants of A* search [HNR68], too. For instance, the 2008 version of Cadiaplayer [FB08] used Memory Enhanced IDA* during the start clock, albeit without a heuristics. If this A* search failed to find a solution during the start clock, Cadiaplayer switches to its Monte-Carlo search method that it also uses for multi-player games (see Section 9.1.3). Méhat and Cazenave also apply Monte-Carlo search to single-player games in [MC10]. Both methods, A* without heuristics and Monte-Carlo search, rely on the game to contain heuristic guidance for the search by giving a part of the full score if only some subgoals are reached. For games that only define the two goal values 100 for winning and 0 for losing, any search algorithm without a heuristic can only find the solution by chance.

Another approach to solving single-player games is described in [Thi09]. This approach takes advantage of the fact that GDL is very similar to Answer Set Programs. In this approach, a GDL game description of a single-player game is transformed into an ASP program such that a model of the program coincides with a solution of the game, that is, a plan for the player to reach a goal state.

A standard ASP system can be used to compute the model. The approach is substantially faster than other search methods for many games but suffers from the same problem as described in Chapter 7 about proving properties of games using ASP: It is only applicable if the game description can be grounded with reasonable memory constraints. A similar method is used in Centurio [MSWS11], a general game player developed at the University of Potsdam.

9.1.2. Multi-player Games with Heuristic Search

Characteristic for heuristic search is the use of a state evaluation function (also called heuristic) to evaluate non-terminal states of the game. Classical search methods, such as, minimax [RN95] or its generalisation max^n [LI86] are often used as heuristic search methods by cutting of the search tree after a certain number of steps. This causes the search tree to have leave nodes that refer to non-terminal states of the game. Since the true value of non-terminal states of a game is usually not known, a heuristic is used to estimate the value of these states.

Variants of minimax have been extensively studied in the literature and applied for playing and solving games, such as, Chess [Mor97] and Checkers [SBB+07], successfully. Standard minimax is only applicable to two-player zero-sum games with alternating moves. The max^n algorithm is a generalization of minimax to n-player games with alternating moves. There are several enhancements for Minimax that prune the search space:

Transposition tables A transposition table [Sch89] is a hash map in which explored states are saved. States that are stored in the transposition table do not need to be explored again if they are reached with a different sequence of actions.

Alpha-beta pruning Alpha-beta pruning [RN95] is a method to cut-off the search of moves that are not better than previously explored alternatives (see Section 4.2.2).

History heuristics History heuristics [Sch89] can be used in conjunction with iterative deepening search. The heuristics determines the order in which the successor states of a state are expanded based on the values of the successor states in the previous iteration of the iterative deepening search. This enhancement leads to more cut-offs by the alpha-beta pruning because it increases the chance that good moves are explored earlier.

To take advantage of Alpha-beta pruning some general game players reduce arbitrary n-player games to two-player zero-sum games [Clu08] by making the paranoid assumption [SK00] that all opponents of the player form a coalition with the goal of reducing the player's reward.

The algorithms we discussed above are only applicable to turn-taking games, i. e., games were in every state only one of the players can choose between several moves. In general, games in GGP are modelled as simultaneous move games. A simultaneous move game can, in principle, be solved by computing the Nash equilibria [Nas50] of the game, i. e., a set of strategies for each player

such that no player can increase his reward by unilaterally deviating from his strategy. However, this approach is not used in any successful GGP system to date because of two reasons:

- There is no efficient algorithm to compute Nash equilibria in the n-player case (see, e. g., [DGP06] for the 4-player case).
- There might be several Nash equilibria in a game with different rewards. In this case, it is unclear which of the equilibria, i. e., which strategy, should be chosen by a player. Thus, Nash equilibria alone are not enough to decide which action to choose.

Because of these problems, none of the current general game players that use heuristic search employ Nash equilibria for decision making. Instead, simultaneous move games are usually modelled as alternating moves games by assuming an order in which players select their actions (cf. Section 4.2.3). By doing this transformation, standard minimax or max^n search can be used for the game, albeit with an error: All players except the first, are overestimated, that is, they are modelled stronger than they are in reality. Thus max^n search will not compute the exact value of a state in general, if applied to simultaneous move games.

9.1.3. Multi-player Games with Monte-Carlo Search

While heuristic search needs a state evaluation function to evaluate non-terminal states of the game, Monte-Carlo search methods learn an evaluation of intermediate states by observing the outcome of random simulations of the game.

Monte-Carlo (MC) search methods, especially together with **U**pper **C**onfidence bounds applied to **T**rees (UCT) [KS06], were successfully applied to computer Go [GWMT06, Cou07], a game for which no good heuristics were found so far. This led to high popularity of MC/UCT for GGP starting with Cadiaplayer [FB08] in 2007. Almost all current GGP players use some form of Monte-Carlo tree search or UCT [FB11, MSWS11, MC11, KE11].

In addition to the independence on constructing evaluation functions, Monte-Carlo search methods have the advantage of being easy to run in parallel on many machines [MC11, MSWS11, KE11]. However, further progress with MC/UCT seems to be only possible by introducing game knowledge in the form of heuristic guidance for the random simulations [SKG08, FB10, KSS11]. The methods developed in this thesis can provide this knowledge. Therefore, we suggest to combine the knowledge-based approach developed in this thesis with Monte-Carlo search methods for future work.

9.2. Heuristics

In contrast to heuristics for specific games, evaluation functions in GGP must be either applicable to all games or automatically constructed for the specific game at hand. We will look into heuristics that are used in other GGP systems

in Section 9.2.1. All GGP systems that use heuristics also use some form of automatic feature construction to generate features that are than used as the atomic components of an heuristics. We will discuss other work on feature construction outside of GGP in Section 9.2.2. Finally, we will review domain independent heuristics from the automated planning area in Section 9.2.3.

9.2.1. Heuristics in Other GGP Systems

Several other general game players use heuristic search to play games. We will discuss these in the following.

Metagamer

Barney Pell's Metagamer [Pel93, Pel96] is a general game playing system for symmetric chess-like games. In contrast to today's GGP systems, Metagamer does not use GDL but a more restricted game description language called Metagame [Pel92]. The class of games is restricted to two-player perfect information games where both players move pieces around on a rectangular board. Goals of the game can be either

- depriving the opponent of legal moves,
- capturing all pieces of a certain type,
- reaching a certain position on the board, or
- a disjunctive combination of any of the above.

Metagamer constructs its own state evaluation function from a variety of features, such as:

Mobility Mobility is the number of moves that are available in a state. Metagamer distinguishes between different kinds of mobility, e. g., mobility of certain kinds of pieces or the number of capture moves available.

Threats Threat features measure the value capture moves that are currently available by the value of the piece that would be captured.

Progress Progress measured, e.g., by counting the number of captured pieces, the distance of a piece to the goal location, or the distance to a location in which a piece could get promoted. Which of these measures applies depends on the goal of the game.

Material Value Material value encompasses a whole list of different features that all assign a value to the pieces that are still on the board. These values are based on properties of the piece, such as,

- static mobility of a piece, i. e., the average number of moves a piece can do on an empty board,
- how many other pieces this piece can capture,
- by how many other pieces the can be captured, or
- the value of the pieces a piece can promote into.

As we can see, most of these features only apply to chess-like games, i. e., games with pieces that move on a board. Many of the features even refer to advanced concepts such as, capturing or promotion of a piece. The restricted description language explicitly defines pieces, board locations and move options of pieces. Thus, detecting pieces and boards, as we do it in Chapter 5, is not necessary. Although Barney Pell's research was ground-breaking for GGP, most of the heuristics of Metagamer can not be used in today's GGP systems. Large parts of Metagamer are only applicable for a restricted class of games.

Cluneplayer

Most notable is James Clune's Cluneplayer [Clu07, Clu08], the winner of the first international GGP competition in 2005. Cluneplayer's heuristic function is composed of the following five components:

Payoff The payoff function estimates the payoff a role will get in the game starting in state in the current state. The payoff function is a constructed as a linear combination of features. The features are extracted from the game rules and weights are learned as a correlation of the feature values to actual payoffs in specially constructed artificial game states. We will explain the feature discovery in more detail below.

Mobility The mobility function estimates the relative mobility of a role at the current state, that is, the number of possible moves of the role relative to those of the other roles. However, Cluneplayer does not use the immediate mobility, i. e., the actual number of legal moves in the state, as this might be misleading in games with alternating moves or zugzwang. Instead, mobility is computed as a combination of features similar that are relevant to the legality of moves. Furthermore, only those features are used that are considered stable, that is, whose variance from one state to the next is not too high. This ensures that the value of the heuristics does not oscillate wildly.

Stability of payoff and stability of mobility Cluneplayer measures stability of the payoff and mobility functions by statistical properties of the values of these functions over sample matches. The intuition behind these two values is that quantities that oscillate wildly do typically not provide as good a basis for an evaluation function as quantities that only vary incrementally.

Termination The termination function estimates how close the current state is to a terminal state. Termination is a linear function on the number of steps executed so far. The parameters of this linear function are learned from sample matches by least squares regression.

These five components are combined into a heuristic function in such a way that the mobility is more important at the start of the game and payoff has more influence towards the end of the game. The progress of the game is estimated by the termination function. Both, payoff and mobility are also regarded less important, if their respective stability is low.

The payoff and mobility functions are constructed as linear combinations of features that Cluneplayer generates in the following way. First, Cluneplayer extracts sub-formulas from the goal condition (for features used in the payoff function) or the legal rules (for features used in the mobility function). Features are than constructed by imposing one of the following interpretations on these formulas:

Solution cardinality Solution cardinality counts the number of ground instances of the variables in a formula that fulfil the formula in the current state. This interpretation is a generalization of our evaluation function for quantified formulas (see Definition 5.2 in Chapter 5). In our evaluation function, we only check whether there is a solution. Computing the solution cardinality is more expensive, because it requires to compute all instead of just one solution. Whether this additional effort pays off in the form of better heuristics needs to be analysed.

Symbol distance Symbol distance uses a distance measure on the terms appearing in the formula, if such a distance measure could be identified in the game. This interpretation is similar to the distance functions for game boards and quantities that we defined in Section 5.3.

Partial solution This interpretation is applied to expressions that are conjunctions or disjunctions and counts how many of the sub-expressions are satisfied. "Partial solution" is a restricted form of our state evaluation function described in Section 5.1 that only applies to conjunctions and disjunctions and uses a binary evaluation of each conjunct or disjunct, respectively.

Cluneplayer's use of mobility as a general heuristics for all games seems to work well. However, there can be games where mobility is a misleading heuristics. To handle those cases well, it would be necessary to automatically detect whether or not mobility applies to each specific game. This is an open problem. We only use mobility indirectly, e. g., in games that end if some player has no legal move left. In those cases, our evaluation function is similar to a mobility heuristics (cf. 5.4).

Clune's concept of stability is interesting for filtering out parts of the heuristics that exhibit erratic behaviour. However, at the moment, it is unclear if and how this concept can be incorporated in our evaluation function. Furthermore, to estimate the stability of features, Cluneplayer uses random matches. However, actual matches are typically not random. The influence of the choice and number of sample matches on the quality of the stability estimation has not been researched.

UTexas/KuhlPlayer

The player named UTexas or KuhlPlayer [KDS06, Kuh10] is another player using heuristic search. Kuhlplayer uses various methods to find structures in the game such as successor relations, step counters, boards, pieces and quantities similar to the methods described in Chapter 5. In contrast to our methods,

which are mostly based on the semantics of game rules, Kuhlplayer uses pattern matching on the syntactic structure of the rules or random simulations of the game to find these structures. This makes Kuhlplayer's approach less general. A number of features are generated from these structures:

- x and y coordinates of each piece,
- Manhattan distance between each pair of pieces,
- number of pieces of each type, and
- the (numeric) value of each quantity.

Opposed to our evaluation function presented in Chapter 5, these features are not combined to an evaluation function for the game. Instead, KuhlPlayer uses a massive parallel approach, in which a number of slaves search the game tree – each with a different feature used as an evaluation function. A majority vote between these slaves is used to decide on the best move.

In his thesis [Kuh10], Gregory Kuhlmann discusses methods to learn complex evaluation functions from a set of features but comes to the conclusion that "the key problem that remains [...] is to identify ways to refine the set of constructed features to a set that is manageable for learning". Learning evaluation functions from a large set of generated features is an open problem in GGP.

In addition to the generation of evaluation functions, KuhlPlayer contains a method to transfer knowledge from one game to similar games [Kuh10]. This knowledge may include features found for the game, importance of the features, but also learned evaluation functions. Knowledge transfer in a general game playing context was previously studied in [BKS06] and [BS07]. Knowledge transfer can greatly reduce the effort for game analysis and learning of evaluation functions. However, knowledge transfer requires to compute the similarity of games, which is intractable, in general.

Knowledge transfer in its current form is of little use to our approach. All the knowledge that we generate is either generated fast (e. g., the evaluation function) or is very specific for the game at hand (e. g., proved properties). Thus, transferring this knowledge to other games is either not worth the effort or not easily possible without losing correctness. The usefulness of knowledge transfer is probably higher for players that use expensive methods (e. g., learning) to generate heuristics or other inexact information about the game. Thus, its application to Monte-Carlo simulation based players that use learned heuristics (e. g., [FB10]) should be studied.

Goblin/Ogre

The players called Goblin and Ogre were developed by David Kaiser [Kai07a, Kai07b, Kai07c]. His players analyse the game by random simulations to detect board structures and pieces and constructs the following features for board games:

- Manhattan distance of a piece from the initial location,

- Manhattan distance of a piece to the goal location,
- number of pieces of each type, and
- "occupied columns", that is, the number of pieces in the same column.

In addition, he uses the following general features that supposedly do not rely on any of the identified structures:

- mobility, i. e., the relative number of legal moves of the players in the current state,
- depth of the current state in the game tree,
- reward for the current state, if the game would end immediately,
- "pattern", which compares the current state to a goal state pattern, that is possibly deduced from the game rules,
- "purse", i. e., the value an ordinal property of the game state, e. g., some amount of money or the number of pieces left.

The features that Ogre detects seem to apply mainly to board games. Furthermore, as with Cluneplayer and Kuhlplayer, move patterns of pieces are neglected, that is, the same distance evaluation (Manhattan distance) is used regardless of the type of piece. As far as we know, our distance estimates developed in Chapter 6 are the only method currently available that is actually based on the possible state transitions.

The features that Ogre constructs are than tested separately in games against random players to select the ones that seem beneficial for the current game. However, features are only tested separately. Thus, Ogre ignores features that are only beneficial if used together. Finally, the heuristic evaluation function is the sum of the selected features. This approach solves the issue of learning weights for features by avoiding to give them weights in the first place. Learning weights of the features could possibly improve the evaluation function.

Summary

All of the GGP systems that use heuristic evaluation functions use some form of automatic feature extraction or construction to generate features that are than used as the atomic components of an evaluation function. Many of the features that the three presented systems identify are similar or identical to ones that we use in Fluxplayer. However, our methods are often more general (e. g., based on the semantics of the game rules instead of syntactic structure) or more accurate (distance estimates based on the possible state transitions instead of Manhattan distance).

9.2.2. Feature Construction and Evaluation

All of the GGP systems that use heuristic evaluation functions automatically extract features from the game and use them as atomic components of the

evaluation function. Other works on feature generation for evaluation functions are

- the ELF system [UP98], which expresses features as conjunctions of fluents and is able to learn new features and features weights from samples, and
- GLEM [Bur99], which expresses features as conjunctions of atomic features, but restricts the legal conjunctions by a set of user-defined patterns to keep the number of features handleable. As opposed to ELF, GLEM does not interweave feature construction and weight learning.

Both approaches are not based on a symbolic description of the game as a set of rules as in GDL, but rather require a set of atomic features to be given.

The Zenith system [Faw93, Faw96] uses analysis of the game rules for deriving features. It uses a Prolog-based description language which has some similarity to GDL. A feature in Zenith is an arbitrary formula over the terms of the description language along with a variable list. The value of the feature in a state is defined as the number of unique instances of the variables in the list that satisfy the formula in the state. For example, in the game of Tic-Tac-Toe, the feature $([\texttt{X}, \texttt{Y}], \texttt{true}(\texttt{cell}(\texttt{X}, \texttt{Y}, \texttt{x})))$ would count the number of cells marked with an `x` while the feature $([], \texttt{true}(\texttt{cell}(\texttt{X}, \texttt{Y}, \texttt{blank})))$ has value 1 if there is any blank cell in the current state and 0 otherwise.

Starting from the goal condition, Zenith iteratively develops a set of features through two processes:

Feature Generation applies a set of transformations to the existing features in order to derive new features. The transformations fall into the four classes:
- **Decomposition** For example, splitting a conjunction in two features.
- **Abstraction** For example, removing a conjunct from a conjunction or a variable from the variable list.
- **Specialisation** For example, instantiation of a variable or removing of a disjunction from a disjunction.
- **Goal Regression** Regression of a formula, that is, replacing each fluent in a formula by its precondition.

Feature Selection The purpose of the feature selection phase is to select the set of features with the best predictive value under the constraint that the overall computation time of all features must not exceed a fixed threshold. In order to measure the predictive value of a feature, Zenith builds an evaluation function from all newly generated features and learns weights of the features by observing state preferences in expert matches. In an iterative process, features with low weights are removed until overall computation time falls below the threshold.

Martin Günther adapted Fawcett's work to the GGP domain in his diploma thesis [Gün08] and successfully learned good evaluation functions for several games. However, the usefulness of the approach is limited by two facts:

- The quality of the learned heuristics is depending on the availability of good samples. However, expert matches or expert reference players do typically not exist for previously unknown games.

- Learning the feature weights in the selection phase is expensive. Thus, this approach can only be used for offline learning of evaluation functions and is of limited use in a competition setting with a restricted start clock.

9.2.3. Heuristics in Automated Planning

As discussed in Section 9.1.1, a direct application of methods used in planning to GGP is hindered by the difference in representation languages. However, several of the planning systems employ heuristic search, e. g., the Heuristic Search Planner (HSP) [BG01] and Fast-Forward (FF) [HN01]. In this section we want to analyse if these heuristics can be used in GGP.

The heuristics used in planning systems such as FF and HSP is an approximation of the plan length of a solution in a relaxed problem, where negative effects of actions are ignored. This heuristics is known as delete list relaxation. While, on first glance, this may easily be applicable to GGP, several problems exist:

- While goal conditions of most planning problems are simple conjunctions, goals in general games can be very complex (e.g., checkmate in Chess). Additionally, the plan length is usually not a good heuristics, given that we can only control our own actions but not those of the opponents. Thus, distance estimates in GGP are usually not used as the only heuristics but only as a feature in a more complex evaluation function (see Chapter 6). As a consequence, computing distance estimates must be relatively cheap because further time is needed for the evaluation of other features.
- Computing the plan length of the relaxed planning problem is NP-hard, and even the approximations used in HSP or FF that are not NP-hard require to search the state space of the relaxed problem.
- The retention of old state properties (because negative effects are ignored) seems to alter general games to a greater extend than it does with planning problems. For example, consider the game Tic-Tac-Toe (see Figure 2.1) in which all negative effects are ignored. In this game, the `mark` action results in cells staying blank in addition to holding the respective marker in the successor state. In contrast to the original game rules, this would effectively render blocking of lines of the opponent useless because the opponent could still mark the cell with his own marker (the cell being blank is the precondition of the mark action). However, depriving the opponent of (good) moves is an essential strategy in many games. With the relaxed problem, this strategy cannot be found anymore.

9.3. Summary

In this chapter, we reviewed previous work on General Game Playing. We especially focused on search algorithms and heuristics, as these topics are most relevant for a knowledge-based general game player. We studied the approaches of

other knowledge-based GGP systems and analysed the applicability of approaches for automated planning to GGP.

10. Discussion

This chapter summarizes the contributions of this thesis and provides a brief outline of possible future work. Furthermore, we list publications that resulted from the research presented in this thesis.

10.1. Contributions

Our main contributions are:

Semantics of the Game Description Language (GDL) Games in the general game playing context are often thought of as state machines where states correspond to the positions of the game and state transitions correspond to the actions that players execute. In Section 2.14 we define a formal semantics of the game description language GDL as a state transition system. This semantics is used in proving the correctness of certain game analysis algorithms, e. g., proving of game properties (Chapter 7) and symmetry detection (Chapter 8).

State Evaluation Function We developed a method to construct effective state evaluation functions for general games (Chapter 5). Our approach solves two non-trivial problems: automatically finding relevant features of a game and learning an evaluation function based on these features. Previous solutions to both problems required input from domain experts [Bur99], expensive learning algorithms [UP98] or both [Faw96]. Our state evaluation function is directly constructed from the rules of the game and does not require learning of weights. It can easily be improved by incorporating further knowledge about the game as we demonstrated in Section 5.3.

Automatic Discovery of Game Structures In Section 5.2, we developed methods to automatically find structures in games, such as, game boards, quantities, and order relations. The goal was to find structures that give rise to a more informed evaluation, such as a distance estimate, in contrast to the basic boolean evaluation that is given by the standard semantics of the game rules. Our algorithms discover these structures based on syntactic and semantic properties of game rules. Thus, the algorithms scale well to complex games because they do not depend on the size of the state space of the game but only on the rules of the game, which are typically exponentially smaller. We showed how the discovered structures can be used to improve the quality of state evaluation functions (Section 5.3).

Distance Estimates In Chapter 6, we presented an algorithm to compute admissible estimates for the number of steps needed to fulfil atomic game

properties, i.e., fluents of the game. We showed how these distance estimates can be used as features in a state evaluation function in the same way as the distance functions defined for the game structures above. In fact, the distance estimates can be seen as a generalization of the distance functions defined for game boards and quantities.

Again the algorithm to compute distance estimates is based on the game rules and is therefore independent on the size of the state space.

Proving Properties More knowledge about the game can be extracted by hypothesising and proving state invariants. In Chapter 7, we present a method for this, based on Answer Set Programming. This method enables us to prove invariants of game states that could not be determined with certainty with the methods presented before, for example, the input and output arguments of fluents (defined in Section 5.2.3) that are needed to detect boards and pieces. Furthermore, the method allows us to prove other game properties that are of interest for deciding which search algorithm is applicable. For instance, if the game is a zerosum game, we can use minimax search and alpha-beta pruning. The prove method is general enough to also prove other state invariants that might be of interest to a general game player. Some examples are:

- The game is turn-taking, i. e., in every game state at most one player has more than one legal move.
- Two fluents can never hold in the same state. For example, in Chess, a player has only one king, thus, the king cannot be at two places at once. Therefore the two fluents describing different positions of the king cannot hold in the same state.
- Certain positions are not reachable. For example, in Checkers, pieces can only occupy black squares. Thus, all fluents describing that a piece is on a white square can never hold in a reachable state.

Information like this could, for instance, be used for more efficient internal state representations in a player. For example, if it is known that white squares can not be occupied in Checkers, they can just be removed from the state.

Symmetry Detection Symmetries, such as symmetric roles or symmetries of the board, occur in many games. Exploiting symmetries can greatly reduce the search space as well as the complexity of other game analysis. In Chapter 8, we developed a sound method for detecting and exploiting symmetries in general games. The method is based on the syntactic structure of the game rules. Thus the execution time and memory requirements are independent on the size of the state space. Hence, the method is also applicable to complex games. Furthermore, our symmetry detection mechanism does not require any additional knowledge about the game, such as, knowledge about game boards or pieces. Thus, it can easily be used by any game player, regardless of the kind of search algorithm that it runs or the game analysation methods that it uses.

10.2. Future Work

Incomplete Information Games Current GGP systems are only able to play deterministic games with complete information. This excludes most card-games or games that involve rolling dice because they contain random elements (shuffling of cards, the number shown on a die after rolling) or incomplete information (a player does not see his opponents' cards). In [Thi10], Thielscher developed GDL-II, an extension of GDL to support non-determinism and incomplete information. The next step should be to review the methods that are currently used for GGP and assess their usefulness for the extended setting. We expect many of the methods in this thesis to be directly applicable to GDL-II, e. g., proving properties of games and symmetry detection, because these methods do not depend on the knowledge of a player at a particular state in a game. Other methods might need to be adapted to deal with incomplete information, e. g., the state evaluation function and distance estimates that are both applied to the state that a player observes at a certain point in the game.

Learning Evaluation Functions In this thesis we did not research how we can adapt or refine the evaluation function by learning methods. Learning evaluation functions for general games from scratch seems to be too expensive, at least for the setting of general game playing competitions (see Section 9.2.2). However, starting with an effective evaluation function and refining it by learning methods seems feasible [MT09]. As a future line of research, we suggest to explore methods for learning parameters of the evaluation function, such as weights, as well as structural changes of the evaluation function, such as the addition or removal of features.

Combining the Knowledge-Based Approach with Monte-Carlo Simulations Both the knowledge-based approach, i. e., heuristic search methods, and Monte-Carlo simulations, which do not need any game knowledge, have their advantages in different kinds of games. For the future, it is advisable to find ways for combining both methods in one system.

A starting point is the introduction of heuristic guidance to Monte-Carlo simulations as used in [SKG08], [FB10], or [KSS11]. However, all these systems are learning heuristics from scratch and do not take advantage of the knowledge that knowledge-based approaches discover in games.

Furthermore, search enhancements, such as, alpha-beta pruning have proven effective ways to reduce the search effort. At the moment, these search enhancements are only used by knowledge-based players, which employ heuristic search. The adaptation of these search enhancements to Monte-Carlo tree search should be studied.

Finally, heuristic search methods suffer from a phenomenon called the horizon effect: The search only visits states up to a certain depth in the game tree. Thus, any event that happens later in the game is beyond this search horizon and is not taken into account. Monte-Carlo simulations typically do not suffer

from this problem in the same way. Thus, the incorporation of Monte-Carlo simulations in heuristic search methods could make the latter more robust.

10.3. Publications

We presented the Fluxplayer system including our state evaluation function and the detection of game structures in the following publications:

- *"Automatic Construction of a Heuristic Search Function for General Game Playing"*. NRAC workshop, 2007 [ST07a]
- *"Fluxplayer: A Successful General Game Player"*. AAAI conference, 2007 [ST07b]

The semantics of the game description language is published in

- *"A Multiagent Semantics for the Game Description Language"*. ICAART conference, 2009 [ST09b]

Our ASP based proof system for game properties is presented in:

- *"Automated Theorem Proving for General Game Playing"*. IJCAI conference, 2009 [ST09a]

The work on symmetry detection was presented in the following publications:

- *"Symmetry Detection in General Game Playing"*. GIGA workshop, 2009 [Sch09]
- *"Symmetry Detection in General Game Playing"*. AAAI conference, 2010 [Sch10]

An overview of our whole knowledge-based approach to General Game Playing was published in the following journal:

- *"Knowledge-Based General Game Playing"*. KI Journal, 2011 [HMST11]

Related to knowledge-based General Game Playing, is the factoring or decomposition of games into independent subgames. Solving independent subgames separately greatly reduces the search effort. Martin Günther investigated the decomposition of single-player games [Gün07]. His work was extended by Dengji Zhao for multi-player games [Zha09]. Both works were supervised by me and resulted in the publications:

- *"Factoring General Games"*. GIGA workshop, 2009 [GST09]
- *"Decomposition of Multi-Player Games"*. Australasian Joint Conference on AI, 2009 [ZST09]

Recently, Thielscher extended the game description language to games with incomplete information and random events [Thi10]. We developed an embedding of this extended language into an action language, namely the well-known Situation Calculus, in

- *"Reasoning About General Games Described in GDL-II"* AAAI conference, 2011 [ST11]

A. The Rules of Breakthrough

```
% Breakthrough is a two-player game played on a chess
% board. Each player has two rows of pawns that can
% move one step forward or one diagonal step.
% The opponent's pawns can be captured.
%
% A player wins if he reaches the opposite side of the
% board or if the opponent has no pieces left.
%
% The game ends if one player wins (there is no draw).

% role definition
role(white).
role(black).

% initial state
init(cellholds(1, 1, white)).
init(cellholds(2, 1, white)).
init(cellholds(3, 1, white)).
init(cellholds(4, 1, white)).
init(cellholds(5, 1, white)).
init(cellholds(6, 1, white)).
init(cellholds(7, 1, white)).
init(cellholds(8, 1, white)).
init(cellholds(1, 2, white)).
init(cellholds(2, 2, white)).
init(cellholds(3, 2, white)).
init(cellholds(4, 2, white)).
init(cellholds(5, 2, white)).
init(cellholds(6, 2, white)).
init(cellholds(7, 2, white)).
init(cellholds(8, 2, white)).
init(cellholds(1, 7, black)).
init(cellholds(2, 7, black)).
init(cellholds(3, 7, black)).
init(cellholds(4, 7, black)).
init(cellholds(5, 7, black)).
init(cellholds(6, 7, black)).
init(cellholds(7, 7, black)).
init(cellholds(8, 7, black)).
```

```
init(cellholds(1, 8, black)).
init(cellholds(2, 8, black)).
init(cellholds(3, 8, black)).
init(cellholds(4, 8, black)).
init(cellholds(5, 8, black)).
init(cellholds(6, 8, black)).
init(cellholds(7, 8, black)).
init(cellholds(8, 8, black)).
init(control(white)).

% legal moves
legal(white, move(X, Y1, X, Y2)) :-
        true(control(white)),
        true(cellholds(X, Y1, white)),
        ++(Y1, Y2),
        cellempty(X, Y2).
legal(white, move(X1, Y1, X2, Y2)) :-
        true(control(white)),
        true(cellholds(X1, Y1, white)),
        ++(Y1, Y2),
        ++(X1, X2),
        not true(cellholds(X2, Y2, white)).
legal(white, move(X1, Y1, X2, Y2)) :-
        true(control(white)),
        true(cellholds(X1, Y1, white)),
        ++(Y1, Y2),
        ++(X2, X1),
        not true(cellholds(X2, Y2, white)).
legal(black, move(X, Y1, X, Y2)) :-
        true(control(black)),
        true(cellholds(X, Y1, black)),
        ++(Y2, Y1),
        cellempty(X, Y2).
legal(black, move(X1, Y1, X2, Y2)) :-
        true(control(black)),
        true(cellholds(X1, Y1, black)),
        ++(Y2, Y1),
        ++(X1, X2),
        not true(cellholds(X2, Y2, black)).
legal(black, move(X1, Y1, X2, Y2)) :-
        true(control(black)),
        true(cellholds(X1, Y1, black)),
        ++(Y2, Y1),
        ++(X2, X1),
        not true(cellholds(X2, Y2, black)).


```

```
87 legal(white, noop) :-
88         true(control(black)).
89 legal(black, noop) :-
90         true(control(white)).
91
92 % successor state
93 next(cellholds(X2, Y2, Player)) :-
94         role(Player),
95         does(Player, move(X1, Y1, X2, Y2)).
96 next(cellholds(X3, Y3, State)) :-
97         true(cellholds(X3, Y3, State)),
98         role(Player),
99         does(Player, move(X1, Y1, X2, Y2)),
100        distinctcell(X1, Y1, X3, Y3),
101        distinctcell(X2, Y2, X3, Y3).
102
103 next(control(white)) :-
104        true(control(black)).
105 next(control(black)) :-
106        true(control(white)).
107
108 % terminal conditions
109 terminal :-
110        whitewin.
111 terminal :-
112        blackwin.
113
114 % goal conditions
115 goal(white, 100) :-
116        whitewin.
117 goal(white, 0) :-
118        not whitewin.
119 goal(black, 100) :-
120        blackwin.
121 goal(black, 0) :-
122        not blackwin.
123
124 % auxiliary predicates
125 cell(X, Y) :-
126        index(X),
127        index(Y).
128
129 cellempty(X, Y) :-
130        cell(X, Y),
131        not true(cellholds(X, Y, white)),
132        not true(cellholds(X, Y, black)).
133
```

```
134 distinctcell(X1, Y1, X2, Y2) :-
135         cell(X1, Y1),
136         cell(X2, Y2),
137         distinct(X1, X2).
138 distinctcell(X1, Y1, X2, Y2) :-
139         cell(X1, Y1),
140         cell(X2, Y2),
141         distinct(Y1, Y2).
142
143 whitewin :-
144         index(X),
145         true(cellholds(X, 8, white)).
146 whitewin :-
147         not blackcell.
148
149 blackwin :-
150         index(X),
151         true(cellholds(X, 1, black)).
152 blackwin :-
153         not whitecell.
154
155 whitecell :-
156         cell(X, Y),
157         true(cellholds(X, Y, white)).
158
159 blackcell :-
160         cell(X, Y),
161         true(cellholds(X, Y, black)).
162
163 index(1).
164 index(2).
165 index(3).
166 index(4).
167 index(5).
168 index(6).
169 index(7).
170 index(8).
171
172 ++(1, 2).
173 ++(2, 3).
174 ++(3, 4).
175 ++(4, 5).
176 ++(5, 6).
177 ++(6, 7).
178 ++(7, 8).
```

Bibliography

[ABW87] Krzysztof Apt, H. A. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, chapter 2, pages 89–148. Morgan Kaufmann, 1987.

[ARMS02] Fadi A. Aloul, Arathi Ramani, Igor L. Markov, and Karem A. Sakallah. Solving difficult sat instances in the presence of symmetry. In *Design Automation Conference*. University of Michigan, June 2002.

[BG01] Blai Bonet and Hector Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1-2):5–33, 2001.

[BKS06] Bikramjit Banerjee, Gregory Kuhlmann, and Peter Stone. Value function transfer for general game playing. In *ICML workshop on Structural Knowledge Transfer for Machine Learning*, 2006.

[Bri11] Encyclopædia Britannica. Game. `http://www.britannica.com/EBchecked/topic/224863/game`, January 2011.

[BS07] Bikramjit Banerjee and Peter Stone. General game learning using knowledge transfer. In *The 20th International Joint Conference on Artificial Intelligence*, pages 672–677, 2007.

[Bur99] Michael Buro. From simple features to sophisticated evaluation functions. In *CG '98: Proceedings of the First International Conference on Computers and Games*, pages 126–145, London, UK, 1999. Springer-Verlag.

[Cla78] Keith Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[Clu07] James Clune. Heuristic evaluation functions for general game playing. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1134–1139, Vancouver, July 2007. AAAI Press.

[Clu08] James Clune. *Heuristic Evaluation Functions for General Game Playing*. PhD thesis, University of California, Los Angeles, 2008.

[Cou07] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *Proceedings of the 5th international conference on Computers and games*, CG'06, pages 72–83, Berlin, Heidelberg, 2007. Springer-Verlag.

[DGP06] Constantinos Daskalakis, Paul W. Goldberg, and Christos H. Papadimitriou. The complexity of computing a nash equilibrium.

In *Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*, STOC '06, pages 71–78, New York, NY, USA, 2006. ACM.

[Elo10] Jarno Elonen. Nanohttpd, 2010. http://elonen.iki.fi/code/nanohttpd/.

[Faw93] Tom E. Fawcett. *Feature Discovery for Problem Solving Systems*. PhD thesis, University of Massachusetts, Amherst, 1993.

[Faw96] Tom E. Fawcett. Knowledge-based feature discovery for evaluation functions. *Computational Intelligence*, 12(1):42–64, 1996.

[FB08] Hilmar Finnsson and Yngvi Björnsson. Simulation-based approach to general game playing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 259–264, Chicago, July 2008. AAAI Press.

[FB10] Hilmar Finnsson and Yngvi Björnsson. Learning simulation control in general game playing agents. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 954–959, Atlanta, July 2010. AAAI Press.

[FB11] Hilmar Finnsson and Yngvi Björnsson. Cadiaplayer: Search-control techniques. *KI*, 25(1):9–16, 2011.

[FL99] Maria Fox and Derek Long. The detection and exploitation of symmetry in planning problems. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 956–961, 1999.

[FL02] M. Fox and D. Long. Extending the exploitation of symmetries in planning. In *Proceedings of AIPS'02*, 2002.

[Gel08] Michael Gelfond. Answer sets. In F. van Harmelen, V. Lifschitz, and B. Porter, editors, *Handbook of Knowledge Representation*, pages 285–316. Elsevier, 2008.

[Gen98] M. Genesereth. Knowledge interchange format, 1998.

[GL88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Proceedings of the International Joint Conference and Symposium on Logic Programming (IJCSLP)*, pages 1070–1080, Seattle, OR, 1988. MIT Press.

[GLP05] Michael R. Genesereth, Nathaniel Love, and Barney Pell. General game playing: Overview of the aaai competition. *AI Magazine*, 26(2):62–72, 2005.

[GST09] Martin Günther, Stephan Schiffel, and Michael Thielscher. Factoring general games. In Yngvi Björnsson, Peter Stone, and Michael Thielscher, editors, *Proceedings of the IJCAI-09 Workshop on General Game Playing (GIGA'09)*, pages 27–34, Pasadena, California, USA, July 2009.

[Gün07] Martin Günther. Decomposition of single player games. Großer Beleg, TU-Dresden, 2007.

[Gün08] Martin Günther. Automatic feature construction for general game playing. Master's thesis, Technische Universität Dresden, 2008.

[GWMT06] Sylvain Gelly, Yizao Wang, Rémi Munos, and Olivier Teytaud. Modification of UCT with Patterns in Monte-Carlo Go. Research Report RR-6062, INRIA, 2006.

[HMST11] Sebastian Haufe, Daniel Michulke, Stephan Schiffel, and Michael Thielscher. Knowledge-based general game playing. *KI*, 25(1):25–33, 2011.

[HN01] Jörg Hoffmann and Bernhard Nebel. The ff planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[HNR68] Peter Hart, Nils Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107, 1968.

[Kai07a] David M. Kaiser. Automatic feature extraction for autonomous general game playing agents. In *Proceedings of the Sixth International Joint Conference on Autonomous Agents and Multiagent Systems*, 2007.

[Kai07b] David M. Kaiser. The design and implementation of a successful general game playing agent. In *The Florida AI Research Society Conference*, pages 110–115, 2007.

[Kai07c] David M. Kaiser. *The Structure Of Games*. PhD thesis, Florida International University, Miami, 2007.

[KDS06] Gregory Kuhlmann, Kurt Dresner, and Peter Stone. Automatic heuristic construction in a complete general game player. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence*, pages 1457–62. AAAI Press, July 2006.

[KE09] Peter Kissmann and Stefan Edelkamp. Instantiating general games. In Yngvi Björnsson, Peter Stone, and Michael Thielscher, editors, *Proceedings of the IJCAI-09 Workshop on General Game Playing (GIGA'09)*, Pasadena, California, USA, July 2009.

[KE11] Peter Kissmann and Stefan Edelkamp. Gamer, a general game playing agent. *KI*, 25(1):49–52, 2011.

[KGK95] Rudolf Kruse, Jörg Gebhardt, and Frank Klawonn. *Fuzzy-Systeme*. Teubner, 1995.

[Kor90] Richard E. Korf. Real-time heuristic search. *Artificial Intelligence*, 42(2-3):189–211, 1990.

[KS06] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *In: ECML-06. Number 4212 in LNCS*, pages 282–293. Springer, 2006.

[KS07] Gregory Kuhlmann and Peter Stone. Graph-based domain mapping for transfer learning in general games. In *Proceedings of The European Conference on Machine Learning*, September 2007.

[KSS11] Mesut Kirci, Nathan R. Sturtevant, and Jonathan Schaeffer. A ggp feature learning algorithm. *KI*, 25(1):35–42, 2011.

[Kuh10] Gregory John Kuhlmann. *Automated Domain Analysis and Transfer Learning in General Game Playing.* PhD thesis, University of Texas at Austin, 2010.

[LB08] Kevin Leyton-Brown. *Essentials of Game Theory: A Concise, Multidisciplinary Introduction (Synthesis Lectures on Artificial Intelligence and Machine Learning).* Morgan and Claypool Publishers, 1 edition, June 2008.

[LF03] D. Long and M. Fox. Symmetries in planning problems. In *Proceedings of SymCon'03 (CP Workshop)*, 2003.

[LHH+08] Nathaniel Love, Timothy Hinrichs, David Haley, Erik Schkufza, and Michael Genesereth. General game playing: Game description language specification. Technical Report LG–2006–01, Stanford Logic Group, Computer Science Department, Stanford University, 353 Serra Mall, Stanford, CA 94305, March 2008. Available at: `games.stanford.edu`.

[LI86] Carol Luckhart and Keki B. Irani. An algorithmic solution of n-person games. In *Fifth National Conference of the American Association for Artificial Intelligence (AAAI-86)*, pages 158–162, 1986.

[Llo87] John Lloyd. *Foundations of Logic Programming.* Series Symbolic Computation. Springer, second, extended edition, 1987.

[LPFe10] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, and et.al. Dlv, 2010. http://www.dlvsystem.com/dlvsystem/index.php/DLV.

[LT86] John Lloyd and R. Topor. A basis for deductive database systems II. *Journal of Logic Programming*, 3(1):55–67, 1986.

[LT94] Vladimir Lifschitz and Hudson Turner. Splitting a logic program. In *Principles of Knowledge Representation*, pages 23–37. MIT Press, 1994.

[MC10] Jean Méhat and Tristan Cazenave. Combining uct and nested monte carlo search for single-player general game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):271–277, 2010.

[MC11] Jean Méhat and Tristan Cazenave. A parallel general game player. *KI*, 25(1):43–47, 2011.

[Mcd00] Drew Mcdermott. The 1998 ai planning systems competition. *AI Magazine*, 21:35–55, 2000.

[Mor97] Robert Morris, editor. *Deep Blue Versus Kasparov: The Significance for Artificial Intelligence*. AAAI Press, 1997.

[MSWS11] Maximilian Möller, Marius Schneider, Martin Wegner, and Torsten Schaub. Centurio, a general game player: Parallel, java- and asp-based. *KI*, 25(1):17–24, 2011.

[MT09] Daniel Michulke and Michael Thielscher. Neural networks for state evaluation in general game playing. In *ECML PKDD '09: Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 95–110, Berlin, Heidelberg, 2009. Springer-Verlag.

[Nas50] John F. Nash. Equilibrium points in n-person games. In *Proceedings of the National Academy of Sciences of the United States of America*, 1950.

[NSS99] Ilkka Niemelä, Patrik Simons, and Timo Soininen. Stable model semantics of weight constraint rules. In *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99), volume 1730 of Lecture*, pages 317–331. Springer-Verlag. LNAI, 1999.

[oP10] University of Potsdam. Potassco, the potsdam answer set solving collection, 2010. http://potassco.sourceforge.net/.

[Pel92] Barney Pell. Metagame in symmetric chess-like games. *Heuristic Programming in Artificial Intelligence 3 – The Third Computer Olympiad*, 1992.

[Pel93] Barney Pell. *Strategy generation and evaluation for meta-game playing*. PhD thesis, University of Cambridge, 1993.

[Pel96] Barney Pell. A strategic metagame player for general chess-like games. *Computational Intelligence*, 12:177–198, 1996.

[Pit68] Jacques Pitrat. Realization of a general game-playing program. In *IFIP Congress*, pages 1570–1574, 1968.

[Pug05] Jean-Francois Puget. Automatic detection of variable and value symmetries. In Peter van Beek, editor, *CP*, volume 3709 of *Lecture Notes in Computer Science*, pages 475–489. Springer, 2005.

[Rei89] Alexander Reinefeld. *Spielbaum-Suchverfahren*, volume 200 of *Informatik-Fachberichte*. Springer, 1989.

[RN95] S. Russel and P. Norvig. *Artificial Intelligence*. 1995.

[Rüd09] Christian Rüdiger. Use of existing planners to solve single-player games. Großer Beleg, TU-Dresden, 2009.

[RvdHW09] Ji Ruan, Wiebe van der Hoek, and Michael Wooldridge. Verification of games in the game description language. *Journal of Logic and Computation*, 19:1127–1156, 2009.

[SBB+07] Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Muller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *Science*, pages 1518–1522, July 2007.

[Sch89] J. Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(11), 1989.

[Sch09] Stephan Schiffel. Symmetry detection in general game playing. In Yngvi Björnsson, Peter Stone, and Michael Thielscher, editors, *Proceedings of the IJCAI-09 Workshop on General Game Playing (GIGA'09)*, pages 67–74, Pasadena, California, USA, July 2009.

[Sch10] Stephan Schiffel. Symmetry detection in general game playing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 980–985, Atlanta, July 2010. AAAI Press.

[Sha50] Claude Shannon. Programming a computer for playing chess. *Philosophical Magazine 7*, 41(314):256–275, 1950.

[Sim08] Patrik Simons. Smodels, 2008. http://www.tcs.hut.fi/Software/smodels/.

[SK00] Nathan R. Sturtevant and Richard E. Korf. On pruning techniques for multi-player games. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 201–207. AAAI Press, 2000.

[SKG08] Shiven Sharma, Ziad Kobti, and Scott D. Goodwin. Knowledge generation for improving simulations in uct for general game playing. In *Australian Joint Conference on Artificial Intelligence*, pages 49–55, 2008.

[SLG08] Eric Schkufza, Nathaniel Love, and Michael R. Genesereth. Propositional automata and cell automata: Representational frameworks for discrete dynamic systems. In *Australasian Conference on Artificial Intelligence*. Springer, 2008.

[ST07a] Stephan Schiffel and Michael Thielscher. Automatic construction of a heuristic search function for general game playing. In *Proceedings of the Workshop on Nonmonotonic Reasoning, Action and Change at IJCAI*, Hyderabad, India, January 2007.

[ST07b] Stephan Schiffel and Michael Thielscher. Fluxplayer: A successful general game player. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI-07)*, pages 1191–1196, Vancouver, 2007. AAAI Press.

[ST09a] Stephan Schiffel and Michael Thielscher. Automated theorem proving for general game playing. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2009.

[ST09b] Stephan Schiffel and Michael Thielscher. A multiagent semantics for the game description language. In J. Filipe, A. Fred, and

B. Sharp, editors, *International Conference on Agents and Artificial Intelligence (ICAART)*, Porto, 2009. Springer.

[ST11] Stephan Schiffel and Michael Thielscher. Reasoning about general games described in GDL-II. In *Proceedings of the AAAI Conference on Artificial Intelligence*, San Francisco, August 2011. AAAI Press.

[TF06] J. Tromp and G. Farnebäck. Combinatorics of go. In *Proceedings of 5th International Conference on Computer and Games*, Torino, Italy, May 2006.

[Thi09] Michael Thielscher. Answer set programming for single-player games in general game playing. In P. Hill and D. Warren, editors, *Proceedings of the International Conference on Logic Programming (ICLP)*, volume 5649 of *LNCS*, pages 327–341, Pasadena, July 2009. Springer.

[Thi10] Michael Thielscher. A general game description language for incomplete information games. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 994–999, Atlanta, July 2010. AAAI Press.

[TV10] Michael Thielscher and Sebastian Voigt. A temporal proof system for general game playing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 1000–1005, Atlanta, July 2010. AAAI Press.

[UP98] Paul E. Utgoff and Doina Precup. Constructive function approximation. In Huan Huan Liu and Hiroshi Motoda, editors, *Feature extraction, construction, and selection: A data-mining perspective*, pages 219–235. Kluwer, 1998.

[vG89] A. van Gelder. The alternating fixpoint of logic programs with negation. In *Proceedings of the 8th Symposium on Principles of Database Systems*, pages 1–10. ACM SIGACT-SIGMOD, 1989.

[Zha09] Dengji Zhao. Decomposition of multi-player games. Master's thesis, TU-Dresden, 2009.

[Zob70] Albert L. Zobrist. A new hashing method with application for game playing. Technical Report 88, University of Wisconsin, April 1970.

[ZST09] Dengji Zhao, Stephan Schiffel, and Michael Thielscher. Decomposition of multi-player games. In A. Nicholson and X. Li, editors, *Proceedings of the Australasian Joint Conference on Artificial Intelligence*, volume 5866 of *LNCS*, pages 475–484, Melbourne, December 2009. Springer.